NORTH-HOLLAND

AUTOMATIC FINITE UNFOLDING USING WELL-FOUNDED MEASURES

BERN MARTENS AND DANNY DE SCHREYE*

▷ We elaborate on earlier work proposing general criteria to control unfolding during partial deduction of logic programs. We study several techniques relying on more general and more powerful well-founded orderings. In particular, we extend our framework to incorporate lexicographical priorities between argument positions in a goal. We show that this handles some remaining deficiencies in previous methods. We emphasize the development of fully automatic algorithms for finite unfolding, avoiding the use of ad hoc techniques. Through an extensive formalization, we convey an understanding of the common principles underlying the various algorithms. Finally, we exhibit how our structure-based unfolding framework can be adapted to cope with datalog-like constant manipulating predicates in a satisfactory way. ◁

1. INTRODUCTION

Since its introduction to computer science in the late 1960s and early 1970s, the concept of partial evaluation has developed into the core topic of an entire field of research. Recently, the field seems to have matured to the extent that general textbooks can be written, reference [23] constituting a first publication of this kind.

*B. Martens was supported in part by ESPRIT BRA COMPULOG II, Contract 6810, in part by GOA "Non-Standard Applications of Abstract Interpretation," Belgium, and in part as Senior Research Assistant by the K.U. Leuven Research Council. D. De Schreye is a Research Associate of the Belgian National Fund for Scientific Research.

Address correspondence to Bern Martens, Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium. E-mail: bern@cs.kuleuven.ac.be.

Received May 1994; accepted September 1995.

655 Avenue of the Americas, New York, NY 10010

0743-1066/96/$15.00
SSDI 0743-1066(95)00149-2

Partial evaluation has perhaps been studied most intensively by people working in the area of functional languages, a fact which is reflected in the extensive list of bibliographical references included in [23].

Partial evaluation was imported to the field of logic programming in [24]. Since then, it also has flourished there, its overall development being rather separate from the work within the functional approach. Part of the research has mainly addressed pragmatic issues related to the impurities of Prolog. (References [43, 48, 49] can be mentioned as some examples.) An important topic of study has also been the perspective of eliminating the overhead associated with meta interpreters. (See, e.g., [17, 29, 42, 46, 47]). The subject was put on a firm theoretical basis in [31]. Some current trends in the research can be observed in [26]. They include termination of unfolding, further elaboration of semantical foundations, and applicational studies. Finally, some recent work aims at bridging the gap between the work in the functional and the logical programming communities, including an increased interest in self-application of logic program partial evaluators (see, among others, [21], [28], and [39]). In spite of these efforts, some significant differences remain. And, as Komorowski argues in [25], the main basic concept in logic programming is deduction, rather than evaluation. We therefore take his advice, and since our work is clearly situated in a logic programming context, will use the term "partial *deduction*" rather than "partial evaluation" throughout the rest of this paper.

The basic component of a partial deduction method for logic programs (see, e.g., [2] or [37]) is a procedure computing a partial deduction of an atom A, given an SLD-tree for $\leftarrow A$ with a program P. This requires a method to produce such SLD-trees. Of prime importance is, of course, that they should be *finite* for any A and P. A methodology capable of ensuring this is presented in [8]. It relies on a framework rooted in well-known methods for proving termination of programs, based on well-founded sets, as developed by, e.g., [14] and [32].

In the present paper, we develop further sophisticated techniques relying on the basic framework laid out in [8]. An important source of inspiration for doing so are the many powerful methods developed for proving termination of rewrite systems (see, e.g., [11] and, for a general introduction to rewrite systems, [12]). Well-founded and related orderings also play a major role in that context. Particularly relevant to the work in this paper is the prominent role of lexicographic orderings in the rewrite termination work. However, since there are some important differences between a set of rewrite rules and a logic program, and most of the work in the former context is on *static* termination analysis, we have decided against an attempt to literally translate approaches. Extensive comments on the relationship between rewrite systems and logic programs, seen from the perspective of (static) termination analysis (i.e., given a program and its input, find out whether computation will terminate), can be found in [10].

Of course, providing finite SLD-trees is just one element in the overall setting of developing a good algorithm for partial deduction. Global termination of the entire partial deduction process and the related issue of steering polyvariance (i.e., generating several specialized versions of the same source predicate) are at least equally important problems for which, at present, no general, final solution has been proposed. The interested reader can consult, e.g., [19], [33], or [38] for further comments and results on this topic.

Next, returning to the specific issue of unfolding, it has been observed that ensuring its *termination* does not suffice to provide a good unfolding strategy in

the context of partial deduction. Indeed, unlimited unfolding brings along the danger of *code explosion* (or, to be more precise, *search space explosion*), resulting in a "specialized" program in fact *less* efficient than the original general program. In other words, good unfolding does not only deliver SLD-trees of finite *depth*, but also keeps their *width* within reasonable bounds. While we have not explicitly addressed the latter issue in our current work, the following remarks can be included:

- In principle, the two issues can be considered independently. When finiteness of the trees is guaranteed, extra measures can be added to ensure reasonable width.
- This being said, it can be observed that "data-consumption-based" unfolding, as carried out by the methods proposed here and in [8], already goes a long way towards the latter goal. Since these algorithms only unfold "sufficiently instantiated" atoms, often branching is very low.
- When thoroughly indeterministic programs nevertheless cause wide branching, sophisticated indexing techniques, as available in most Prolog compilers, can efficiently cope with the resulting large number of highly specialized clauses. Again, this specialization is caused by the demand that unfolding should only proceed as long as data are consumed.
- Finally, a medium size experimental study, including a performance comparison of various unfolding methodologies on the benchmark programs proposed in [27], confirms the above observations. It also suggests that only carrying out strictly deterministic unfoldings, as proposed in, e.g., [18], is perhaps a bit too restrictive.

Gallagher mentions a related issue in [19]. Assuming a left-to-right computation rule for logic programs, he points out that unfolding a choice point to the right of another goal can cause duplicated computation and an associated loss of efficiency in the specialized program produced by partial deduction. (See also [24].) Again, this issue, or its generalization to a context without predetermined computation rule, will not be considered in the rest of this paper.

These omissions do not indicate that we believe such considerations to be of minor importance. But, in the present work, we decided to focus on issues in termination of unfolding. We hope that our work contributes to an increased insight in that problem and its possible solutions, and trust that such an understanding will facilitate the further development of good strategies for partial deduction and unfolding.[1] In particular, on the one hand, we propose concrete methods for automatic finite unfolding, extending our earlier work in [8] and [37]. On the other hand, we offer detailed formalizations of the basic ingredients, as well as a generic algorithm, emerging as a common template for automatic unfolding based on the framework in [8]. Finally, we have chosen to structure our presentation in a "bottom-up" way, first elaborating concrete techniques of increasing complexity in ([37, Sect. 3] and) Sections 3 and 4, and later providing a general template in Section 5. In this way, we obtain a smooth transition from our earlier work to the

[1] Good unfolding might, in many cases, be obtained through the use of our methods to produce a (largish) finite SLD-tree (along the way spotting failing branches), and subsequently trimming it to its maximal deterministic subtree. This would provide a very powerful "lookahead" [18]. The matter is the subject of current research.

present paper, and hope to convey a good understanding of both concrete methods and general principles.

This leads to the following layout for this paper. First, in Section 2, we recapitulate and slightly adapt the basic framework laid out in [8]. We also briefly discuss the main characteristics of the automatic unfolding algorithms proposed in [8] and [37]. Section 3 describes a first generalization of the earlier work. More general measure functions are introduced that incorporate lexicographical priorities among arguments in a selected atom. We present an automatic unfolding algorithm relying on such functions, as well as a noteworthy optimization. In Section 4, we build on this work, and expand our horizon to also consider parts of a goal not belonging to the atom to be unfolded, while deciding on unfolding. We show how this enables a treatment of coroutining and better handles the propagation of variable instantiation. Section 5, then, contains a formalization of the underlying issues in the search for optimal measure functions, carried out by automatic unfolding algorithms. Finally, Section 6 forges a unified approach, incorporating both weight-based unfolding and the "checking for a variant ancestor" technique proposed in the literature. It allows a better treatment of predicates that perform heavy constant manipulation. We conclude with a brief overall discussion, rounding off the paper.

Finally, let us mention that a number of the basic ideas underlying the present work are briefly described in [35].

2. WEIGHT-BASED FINITE UNFOLDING

2.1. Introduction

Throughout this paper, we assume familiarity with the basic notions of logic programming (see, e.g., [30]). Moreover, we restrict ourselves to the study of unfolding in (pure) definite programs.

As noted above, the issue of unfolding arises in the context of partial deduction, a formal foundation for which has been laid out in [31]. Following that paper, we extend the notion of SLD-tree by allowing it to be *incomplete*. This means that, apart from success and failure nodes, arbitrary goal statements (where no atom has been selected for further unfolding) also can be leaves. We introduce a piece of useful terminology:

Definition 2.1. A leaf node in an SLD-tree which is neither a success nor a failure node, but an arbitrary goal statement without selected atom, will be called a *dangling* leaf.

We call an SLD-tree *finite* if all its derivations are finite. Observe that for a finite program P, a finite SLD-tree is also a finite tree in the sense that its set of nodes is finite. Indeed, since such a tree is finitely branching, König's lemma (see, e.g., [13]) can be applied.

The following basic definitions on partial deduction are adapted from [31].

Definition 2.2. Let P be a definite program, A an atom, and $\leftarrow A, G_1, \ldots, G_n$ with $n > 0$ an SLD-derivation for $P \cup \{\leftarrow A\}$. Let $\theta_1, \ldots, \theta_n$ be the corresponding sequence of substitutions, and let G_n be $\leftarrow A_1, \ldots, A_m$.

We call $A\theta_1 \cdots \theta_n \leftarrow A_1, \ldots, A_m$ the *resultant* of the derivation $\leftarrow A, G_1, \ldots, G_n$.

Definition 2.3. Let P be a definite program, A an atom, and τ a finite SLD-tree for $P \cup \{\leftarrow A\}$. Let $\{G_i \mid i = 1,\ldots,r\}$ be the (nonroot) leaves of the non-failing branches of τ and $\{R_i \mid i = 1,\ldots,r\}$ the resultants corresponding to the derivations $\{\leftarrow A,\ldots,G_i \mid i = 1,\ldots,r\}$. The set $\{R_i \mid i = 1,\ldots,r\}$ is called a *partial deduction for A in P.*

It can be seen from the above definitions that building a finite, usually incomplete SLD-tree for $P \cup \{\leftarrow A\}$ is indeed a key issue in partial deduction. In previous work, a framework was laid out in which some general methods for that task, of stepwise increasing complexity and power, were described and proved correct. (See [7] and its revised and extended version, [8].) Moreover, [36] and the corresponding extended and revised version, [37], address the issue of sound and complete, termination guaranteed, partial deduction relying on such unfolding methods. As pointed out in Section 1, in the present paper, we will not explicitly consider this partial deduction context, and focus on further issues in automatic finite unfolding.

2.2. The Basic Framework

To keep this paper reasonably self-contained, we now partially recapitulate (sometimes in a slightly adapted version) the material in [8]. Some of the concepts presented below are more general than is strictly necessary in the context of the present paper. We nevertheless decided to include them in order to provide a clear link with our previous work. It goes without saying that the interested reader is invited to consult [8] and [37] for a more complete description of the latter.

Two basic ingredients of our approach are *strict order relations*, denoted $>$, and *well-founded measures*. A strict order relation is an anti-reflexive, anti-symmetric, and transitive binary relation. A (partially) strictly ordered set, $V, >_V$, will be called an *s-poset*, the corresponding order, $>_V$, an *s-order*.

Definition 2.4. An s-poset $V, >$ is called *well-founded* if there is no infinite sequence of elements $e_1, e_2, \ldots$ in V such that $e_i > e_{i+1}$, for all $i \geq 1$.

Definition 2.5. Let $V, >_V$ be an s-poset. A *well-founded measure*, f, *on* $V, >_V$ is a monotonic function from $V, >_V$ to some well-founded set $W, >_W$.

Now, suppose τ is an incomplete SLD-tree, forming part of the complete (possibly infinite) SLD-tree τ_0. Assume a numbering on the nodes of τ_0. Then we can associate with τ the following set: $G_\tau = \{(G, i) \mid G$ is a goal of τ having i as its associated number in $\tau_0\}$. Considering $(G, i) >_\tau (G', j)$ if node i is an ancestor of node j in τ_0 results in $G_\tau, >_\tau$ being an s-poset. This allows the following definition:

Definition 2.6. An SLD-tree τ is *well-founded* if there exists a well-founded measure f on $G_\tau, >_\tau$.

It follows that:

Theorem 2.1. An SLD-tree is finite iff it is well-founded.

However, using Definition 2.6 and Theorem 2.1 as the actual basis for constructing finite SLD-trees is impractical. Indeed, comparing every node in a tree with its

immediate ancestor generally requires very complex measure functions if we want to obtain nontrivially expanded trees. It is usually much more convenient to partition the nodes in a number of sets, and carry out "weight" comparisons setwise. The following has therefore been introduced:

Definition 2.7. An SLD-tree τ is *subsetwise founded* if

1. There exists a finite number of sets, $C_0, \ldots, C_N$, such that $G_\tau = \cup\{C_i \mid i \leq N\}$.
2. For each $i = 1, \ldots, N$, there exists a well-founded measure $f_i : C_i, >_\tau \rightarrow W_i, >_i$.
3. For each $(G, k) \in C_0$ and each derivation D in τ containing (G, k):
   - D is finite or
   - there exists a descendant (G', j) of (G, k) in D such that $(G', j) \in C_i$ for some $i > 0$.

In C_0, nodes are assembled that we do not wish to compare with other nodes (e.g., goals with a nonrecursive selected atom). Imposing condition 3 on C_0 assures that this can be done safely. We have the following theorem:

Theorem 2.2. An SLD-tree is finite iff it is subsetwise founded.

Reference [8] then goes on to show how, indeed, the latter definition and theorem can serve as a basis for ensuring the termination of unfolding. Several methods of increasing power are introduced. In the context of the present paper, we will only briefly exhibit the most complex one. Since it is also the most powerful, it will be used as the starting point and reference for further developments. For details, examples, and ample comments on the underlying intuitions, we refer to [8].

In the sequel, with slight abuse of notation, we will occasionally refer to pairs (G, i) as goals or nodes in τ or τ_0 (instead of G_τ or G_{τ_0}). We will denote by $R(G, i)$ the atom selected in a given goal (G, i) in an SLD-tree τ by the computation rule R used to construct τ. Note, however, that in the context of an incomplete SLD-tree τ, $R(G, i)$ is not defined when (G, i) is a dangling leaf (nor, of course, when it is a success node $(\square, i)$).

Definition 2.8. Let $(G, i) = ((\leftarrow A_1, \ldots, A_j, \ldots, A_n), i)$ be a node in an SLD-tree τ, let $R(G, i) = A_j$ be the atom selected by the computation rule R, let $H \leftarrow B_1, \ldots, B_m$ be a clause whose head unifies with A_j, and let $\theta = mgu(A_j, H)$ be their most general unifier. Then (G, i) has a son (G', k) in the SLD-tree τ, $(G', k) = ((\leftarrow A_1, \ldots, A_{j-1}, B_1, \ldots, B_m, A_{j+1}, \ldots, A_n)\theta, k)$. Let (G'', l) be a descendant of (G', k) in τ with $R(G'', l) = B_r\theta\psi$, for some $r \leq m$ and ψ the composition of all mgus on the subderivation from (G', k) to (G'', l). We say that $B_r\theta\psi$ in (G'', l) is a *direct descendant* of A_j in (G, i), and that A_j in (G, i) is a *direct ancestor* of $B_r\theta\psi$ in (G'', l).

The binary relations *descendant* and *ancestor*, defined on (selected) atoms in goals, are the transitive closures of the direct descendant and direct ancestor relations, respectively. For A an atom in (G, i) and B an atom in (G', j), A is an ancestor of B is denoted as $A >_{pr} B$ ("pr" stands for proof tree).

The relations descendant and ancestor on pairs of atoms in goals of τ induce in a natural way refined descendant and ancestor relations on τ. Let (G, i) and (G', k) be in τ. We call (G, i) a *proper ancestor* of (G', k) if $(G, i) >_\tau (G', k)$ (i.e.,

node i is an ancestor of node k) and $R(G,i) >_{pr} R(G',k)$. Abusing notation, we denote the latter relation between *goals* as $(G,i) >_{pr} (G',k)$.

The following definition describes a key notion in our approach.

Definition 2.9. A (possibly infinite) pair $((C_0, C_1, C_2, \ldots), (f_1, f_2, \ldots))$ is a *hierarchical prefounding* for (the complete, but possibly infinite SLD-tree) τ_0 if:

1. There exists a finite partition $R_0, \ldots, R_N$ of $R_{\tau_0} = \{R(G,i) \mid (G,i) \in \tau_0\}$ such that:
   - $C_i = \{(G,i)\} \cup \{(G',j) \in \tau_0 \mid \exists k : 1 \leq k \leq N$, such that $R(G,i), R(G',j) \in R_k$ and $(G,i) >_{pr} (G',j)$ or $(G',j) >_{pr} (G,i)\}$
   - $C_0 = \{(G,i) \in \tau_0 \mid R(G,i) \in R_0\} \cup \{(\Box, i) \in \tau_0\}$
2. $f_1, f_2, \ldots$ are functions mapping, respectively, $C_1, C_2, \ldots$ to one of a finite number of well-founded sets $W_1, >_1, \ldots, W_N, >_N$ such that f_i maps C_i to W_k if the selected atoms of the goals in C_i belong to R_k. Moreover, for all $i, j > 0 : f_i|_{C_i \cap C_j} = f_j|_{C_i \cap C_j}$.
3. C_0 contains no infinite sequence $(G_{i_1}, i_1) >_{pr} (G_{i_2}, i_2) >_{pr} \ldots$ such that for all $m \geq 1$, $R(G_{i_m}, i_m)$ in (G_{i_m}, i_m) is a direct ancestor of $R(G_{i_{m+1}}, i_{m+1})$ in $(G_{i_{m+1}}, i_{m+1})$.

We need one more definition before we can state a basic theorem obtained in [8].

Definition 2.10. Suppose τ is an SLD-tree. Then we call the tree τ^-, obtained by deleting from τ its dangling leaves, the *SLD$^-$-tree associated to* τ.

Definition 2.10 is more specific than its counterpart in [8]. Notice that the definition given here implies a one-to-one correspondence between SLD-trees and associated SLD$^-$-trees. Definition 2.7 applies to SLD$^-$-trees in the obvious way, and we can state the following theorem:

Theorem 2.3. Let τ be an SLD$^-$-subtree of the complete SLD-tree τ_0, and suppose that $((C_0, C_1, C_2, \ldots), (f_1, f_2, \ldots))$ is a hierarchical prefounding for τ_0. If each f_i is a well-founded measure on $C_i \cap \tau, >_\tau$, then there exists a finite number of sets $C_{i_1}, \ldots, C_{i_M}$ in $\{C_1, C_2, \ldots\}$ such that τ is subsetwise founded with respect to $((C_0 \cap \tau, C_{i_1} \cap \tau, \ldots, C_{i_M} \cap \tau), (f_{i_1}, \ldots, f_{i_M}))$.

Proposition 2.1. Let τ be an SLD-tree and τ^- its associated SLD$^-$-tree. If τ^- is subsetwise founded with respect to $((C_0, C_1, \ldots, C_N), (f_1, \ldots, f_N))$, then τ is subsetwise founded with respect to $((C_0', C_1, \ldots, C_N), (f_1, \ldots, f_N))$, where $C_0' = C_0 \cup \{(G,i) \in \tau \mid (G,i)$ is a dangling leaf of $\tau\}$.

PROOF. All three conditions of Definition 2.7 are immediately verifiable. □

Corollary 2.1. Let τ be an SLD-subtree of the complete SLD-tree τ_0, and suppose that $((C_0, C_1, C_2, \ldots), (f_1, f_2, \ldots))$ is a hierarchical prefounding for τ_0. If each f_i is a well-founded measure on $C_i \cap \tau^-, >_{\tau^-}$, then there exists a finite number of sets $C_{i_1}, \ldots, C_{i_M}$ in $\{C_1, C_2, \ldots\}$ such that τ is subsetwise founded with respect to

$((C_0', C_{i_1} \cap \tau^-, \ldots, C_{i_M} \cap \tau^-), (f_{i_1}, \ldots, f_{i_M}))$ *where* $C_0' = (C_0 \cap \tau^-) \cup \{(G,i) \in \tau \mid (G,i)$ *is a dangling leaf of* $\tau\}$.

PROOF. The corollary follows immediately from Theorem 2.3 and Proposition 2.1. □

Definitions 2.7 and 2.9, Theorem 2.2, and Corollary 2.1 together provide a suitable basis for the construction of algorithms that build large, sensibly expanded, yet finite, incomplete SLD-trees. We now proceed to exhibit (a slightly adapted version of) one such algorithm from [8]. Two more definitions are needed.

Definition 2.11. Let $((C_0, C_1, C_2, \ldots), (f_1, f_2, \ldots))$ be a hierarchical prefounding for τ_0, with associated partition $R_0, \ldots, R_N$ of R_{τ_0}, and let (G,i) and (G',j) be in $\tau_0 \setminus C_0$. We say that (G',j) *covers* (G,i) if the following two conditions are satisfied:

1. $(G',j) >_{pr} (G,i)$
2. $\exists k : R(G',j), R(G,i) \in R_k$.

Note that, in Definition 2.9, a C_i-set ($i \geq 1$) contains precisely all nodes that either cover or are covered by the corresponding (G,i) goal node.

Definition 2.12. Let (G,i) and (G',j) be two distinct nodes of $\tau_0 \setminus C_0$. (G',j) is called the *direct covering ancestor* of (G,i) if:

1. (G',j) covers (G,i) and
2. any other (G'',k) that covers (G,i) also covers (G',j).

It follows that the direct covering ancestor of a node, if it exists, is unique. Finally, note that a hierarchical prefounding of a complete SLD-tree τ_0 is uniquely determined by its associated pair $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$ where:

- $R_0, \ldots, R_N$ is the partition of R_{τ_0} mentioned in the first condition of Definition 2.9
- $F_k : \{(G,i) \in \tau_0 \mid R(G,i) \in R_k\} \rightarrow W_k, >_k$ coincides with f_i on C_i if the selected atoms of the goals in C_i are in R_k

Based on this observation and Corollary 2.1, we will occasionally slightly abuse terminology and call an SLD-tree τ subsetwise founded with respect to a pair $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$.

In the following algorithm, we suppose that a computation rule R (and therefore a complete SLD-tree τ_0) and a pair $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$ satisfying Definition 2.9 are given.

Algorithm 2.1

Initialization
  $\tau := \{\{(\leftarrow A, 1)\}\}$ { * *an SLD-tree with a single one-node derivation* *}
  $Pr := \emptyset$ { * *in* Pr, *the* $>_{pr}$*-relation will be constructed* *}
  Terminated $:= \emptyset$
While *there exists a derivation* $D \in \tau$ *such that* $D \notin$ *Terminated* **do**
  Let (G,i) *be the leaf of* D
  Let Resolvents(G,i) *be the set of all its direct* $>_{\tau_0}$*-descendants*

If *Resolvents*$(G,i) = \emptyset$ {* (G,i) *is a success or a failure node* *}
Then *add* D *to Terminated*
Else if *there is a direct covering ancestor* (G',j) of (G,i) *with* $R(G',j)$, $R(G,i) \in R_n$ *such that not*$(F_n(G',j) >_n F_n(G,i))$
Then *add* D *to Terminated* {* (G,i) *becomes a dangling leaf* *}
Else {* τ *is further extended* *}
Replace τ *by* $\tau \setminus D \cup \{D \cup \{(G'',k)\} \mid (G'',k) \in$ *Resolvents*$(G,i)\}$
Extend the Pr-relation accordingly
Endwhile

In words, the operation of the above algorithm can be sketched as follows. Unfolding is controlled by a number of "measure" functions that are kept monotonically decreasing on certain subsets of goals in the tree. Termination of the algorithm (and thus finiteness of the resulting SLD-tree) is assured by having well-founded sets as the ranges of the measure functions, and making sure that each derivation contains only a finite number of goals for which no comparison with an ancestor goal is imposed. Within each derivation, goals are compared when their selected atoms belong to the same of a number of given classes, and one is a descendant of the other. For further details and examples, we refer to [8]. We have the following results:

Proposition 2.2. Let $((C_0, C_1, C_2, \ldots), (f_1, f_2, \ldots))$ *be the hierarchical prefounding of the complete SLD-tree* τ_0, *determined by the pair* $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$ *and the computation rule* R, *underlying an application of Algorithm 2.1. Let* τ *be the SLD-tree built by Algorithm 2.1. Then there exists a finite number of sets* $C_{i_1}, \ldots, C_{i_M}$ *in* $\{C_1, C_2, \ldots\}$ *such that* τ *is subsetwise founded with respect to* $((C_0', C_{i_1} \cap \tau^-, \ldots, C_{i_M} \cap \tau^-), (f_{i_1}, \ldots, f_{i_M}))$ *where* $C_0' = (C_0 \cap \tau^-) \cup \{(G,i) \in \tau \mid (G,i)$ *is a dangling leaf of* $\tau\}$.

Theorem 2.4. Algorithm 2.1 terminates. The resulting SLD-tree τ *is finite.*

PROOF. This follows from Proposition 2.2 and Theorem 2.2. □

Actually, Algorithm 2.1 differs slightly from Algorithm 3.2 in [8]. Indeed, the approach there does associate a measure to a dangling leaf, and therefore sometimes (although often not) generates smaller SLD-trees than the method just presented. The issue can be of some importance with respect to the desired amount of specialization in partial deductions, but is a minor one in the restricted context of terminating unfolding. We feel that the present choice allows a more elegant treatment of full automation.

2.3. Automation

Having established the above framework, [8] very briefly addresses some issues related to automation. Indeed, in Algorithm 2.1, the computation rule R, the partition $R_0, R_1, \ldots, R_N$, and the measures $F_1, \ldots, F_N$ are supposed to be given (by the user). A fully automatic unfolding algorithm, however, just takes a definite program P and goal G, and produces a maximally/sensibly expanded, finite SLD-tree for $P \cup G$. One way to proceed towards this goal is sketched in this subsection.

We set out with two more definitions.

Definition 2.13. Let *Term* denote the set of terms in the first-order language used to define the theory P. We define $|.| : Term \rightarrow \mathbb{N}$ as follows:

If $t = f(t_1, \ldots, t_n), n > 0$
then $|t| = 1 + |t_1| + \cdots + |t_n|$
else $|t| = 0$.

Definition 2.14. Let p be a predicate of arity n, and $S = \{a_1, \ldots, a_m\}, 1 \leq a_k \leq n, 1 \leq k \leq m$, a set of argument positions for p. We define $|.|_{p,S} : \{A \mid A$ is an atom with predicate symbol $p\} \rightarrow \mathbb{N}$ as follows:

$$|p(t_1, \ldots, t_n)|_{p,S} = |t_{a_1}| + \cdots + |t_{a_m}|.$$

Next, we fix $R_0, R_1, \ldots, R_N$:

- First, N is taken equal to the number of recursive predicates in P.
- R_0 will contain all selected atoms with a nonrecursive predicate.
- We associate one $R_i (i > 0)$ with each recursive predicate.

For $F_1, \ldots, F_N$, we can take measures as defined in Definition 2.14. Indeed, $\mathbb{N}$ with its standard ordering is a well-founded set. To be precise: we associate with a goal the natural number which is the image of its selected atom under a particular mapping as introduced in Definition 2.14. We will in the sequel often refer to such numbers as *weights*, both of the goal and its selected atom.

It remains to fix a computation rule R, and the exact sets of argument positions S_p to be used for each recursive predicate p. As explained in [8], this can be regarded as a dynamic decision problem. Essentially, we search goals from left to right in order to find an atom which can safely be unfolded. Initially, each S_p is taken to be the full set of argument positions of the predicate p. Elements are removed from S_p dynamically if this operation results in the possibility of building a larger subsetwise founded tree under the new measure.

Automatic unfolding algorithms based on these ingredients appear to give reasonable results for large classes of programs. Experiments in the context of partial deduction can be found in [22] and [33]. However, the same experiments also, not unexpectedly, show that structure-based measures as introduced in Definitions 2.13 and 2.14 can behave poorly in the context of datalog (i.e., function-free) programs. Some other issues not properly dealt with are coroutining and (back)propagation of instantiations. These observations are the starting point for the present paper.

3. LEXICOGRAPHICAL PRIORITIES

3.1. Introduction

In this section, we consider a first generalization of the automatic unfolding methods proposed in [8] and [37]. Those methods rely on measures considering a subset of the selected atom's argument positions, as introduced in Definition 2.14. It turns out that solving some of the problems mentioned above requires measures that also take into account arguments of other atoms in the goal, and moreover, are capable of imposing a priority between different (subsets of) arguments.

A method incorporating these facilities will be presented in the next section. First, we prepare the way by elaborating an intermediary extension of our earlier work. We will, in this section, still limit ourselves to measures based on the arguments of a goal's selected atom, but we will *impose priorities among different argument positions.* We will show that this first advance towards more generality and elegance already results in increased unfolding power.

Below, we first introduce measure functions based on partitions of a predicate's set of argument positions, and show that they can be used as well-founded measures on an SLD-tree, resulting in increased unfolding potential. Next, we present a fully automatic unfolding algorithm, capable of discovering optimal partition-based measures. Finally, we show that an important simplification of the algorithm keeps the refined control and the guaranteed termination while efficiency is improved.

3.2. More Powerful Measures

We set out by introducing the following bits of notation:

- Let V be a set. Then $\mathcal{P}(V)$ denotes the powerset (or set of subsets) of V.
- Let V be a set. Then V^n denotes the n-fold Cartesian product $V \times V \times \cdots \times V$ (n copies) of V.

This allows us to define the following:

Definition 3.1. Let V be a set, and let $S_1, \ldots, S_k$ be k mutually disjoint, nonempty subsets of V, together forming a partition of V. Then the k-tuple $(S_1, \ldots, S_k) \in \mathcal{P}(V)^k$ is called an *ordered k-partition* of V.

In the sequel, we will often simply use the term *ordered partition* when the dimension (k) is clear from the context, or unimportant. Moreover, our attention will focus on ordered partitions of the set of argument positions of predicate symbols. In this context, we will refer to a predicate symbol and an *associated* ordered (k-)partition, without explicit mention of the fact that the latter is a partition of the former's *set of argument positions.*

Definition 3.2. Let p be a predicate of arity n and $O = (\{i_{11}, \ldots, i_{1j}\}, \ldots, \{i_{k1}, \ldots, i_{kl}\})$ an associated ordered k-partition. We define $|.|_{p,O}$: $\{A \mid A \text{ is an atom with predicate symbol } p\} \rightarrow \mathbb{N}^k$ as follows:

$$|p(t_1, \ldots, t_n)|_{p,O} = (|t_{i_{11}}| + \cdots + |t_{i_{1j}}|, \ldots, |t_{i_{k1}}| + \cdots + |t_{i_{kl}}|)$$

where $|.|$ is the term norm as defined in Definition 2.13.

Example 3.1.

$$|p([a,b,c], f(g(a)), b)|_{p,(\{1,2,3\})} = (3 + 2 + 0) = (5)$$
$$|p([a,b,c], g(a), b)|_{p,(\{2\},\{1,3\})} = (1, 3).$$

We intend to use such partition-based measure functions instead of the subset-based ones introduced in Definition 2.14. However, an atom is no longer mapped into an element of $\mathbb{N}$, but into a tuple in $\mathbb{N}^k$. (k, of course, can be 1, as in the first

line of Example 3.1 above.) We must therefore first establish that we can indeed define an order on $\mathbb{N}^k$ such that it becomes a well-founded set.

Definition 3.3. Let V be a set with an order $>$. Then we define the *lexicographical order* $\succ_k$ on V^k as follows:

$$(v_1, \ldots, v_k) \succ_k (w_1, \ldots, w_k)$$
$$\text{iff}$$
$$\forall 1 \leq i \leq k : v_i \equiv w_i \text{ or } v_i > w_i \text{ or } w_i > v_i$$
$$\text{and}$$
$$\exists 1 \leq i \leq k : v_i > w_i \text{ and } \forall 1 \leq j < i : v_j \equiv w_j.$$

Notice that the relation $\succ_k$ thus defined is indeed a strict order relation. The first condition in the above definition is necessary in the context of partial orders to impose the restriction that the components of both tuples should be pairwise comparable. For $\mathbb{N}$ and $\mathbb{N}^k$, this condition is, of course, trivially satisfied since we have a total order on $\mathbb{N}$.

Proposition 3.1. Let $V, >$ be a well-founded set, and $\succ_k$ the associated lexicographical order on V^k. Then $V^k, \succ_k$ is a well-founded set.

PROOF. The proposition follows from the well-foundedness of $V, >$, Definition 3.3, and the fact that a tuple in V^k has only a finite number, k, of components. □

In particular, for each k, $\mathbb{N}^k, \succ_k$ is a well-founded set. This means that functions as introduced in Definition 3.2 can indeed be used as measure functions to control unfolding.

Proposition 3.2. Let O be an ordered k-partition associated to a predicate p. Let τ be an SLD-tree, and S_τ a subset of G_τ such that all goals in S_τ have a selected atom with predicate p. Suppose F_p is defined as follows:

$$F_p : S_\tau, >_\tau \rightarrow \mathbb{N}^k, \succ_k : (G, i) \in S_\tau \mapsto |R(G, i)|_{p,O}$$

Then F_p is a well-founded measure on $S_\tau, >_\tau$ iff F_p is monotonic.

PROOF. This immediately follows from Definition 2.5 and Proposition 3.1. □

Doing so can entail a gain in unfolding capacity, as the following (artificial but simple) example shows.

Example 3.2. Consider the following program:

$$produce\_consume([X \mid Xs], []) \leftarrow produce\_consume(Xs, [X])$$
$$produce\_consume(X, [Y \mid Ys]) \leftarrow produce\_consume(X, Ys)$$

and query:

$$\leftarrow produce\_consume([1, 2 \mid Xs], []).$$

Now, apply Algorithm 2.1 using a single R-class for all selected atoms, with associated measure function $|.|_{p\_c,O}$ where $O = (\{1\}, \{2\})$ is an ordered 2-partition

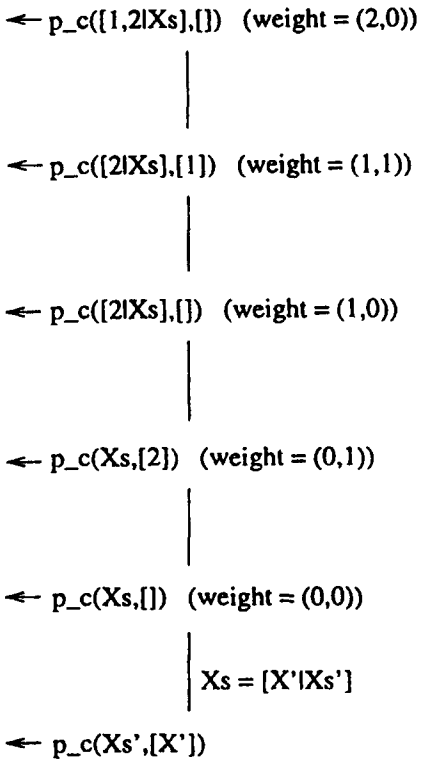

FIGURE 1. Unfolding with weights in $\mathbb{N}^2$.

associated to the predicate *produce_consume*, abbreviated to *p_c*. The resulting incomplete SLD-tree is depicted in Figure 1. Nodes are annotated with their weight, except the last one, which is a dangling leaf.

Notice that, when a simple one-component weight of the kind introduced in Definition 2.14 is used, only a trivial single step is possible if both arguments (or only the second) are considered, while focusing on the first argument causes termination after two steps.

3.3. An Automatic Unfolding Algorithm

In this subsection, we present a detailed, fully automatic algorithm for unfolding, based on the ingredients introduced above. It is a first enhancement of the algorithms described in [8] and [37]. We would like to obtain a concise and clear formulation of the algorithm, including the automatic search for optimal measure functions. To make this possible, we first include some helpful definitions and prove a few relevant properties about the concepts they introduce.

3.3.1. Setting the Scene. We set out with some straightforward definitions related to the behavior of measure functions on a pair of atoms.

Definition 3.4. Let p be a predicate of arity n. Let $P_1 = p(t_1, \ldots, t_n)$ and $P_2 = p(s_1, \ldots, s_n)$ be two atoms, and F a mapping from the set of terms in the language underlying P_1 and P_2 to an s-poset $V, >$. Then an *argument position* i $(1 \leq i \leq n)$ is:

- (P_1, P_2)-*decreasing* for F iff $F(t_i) > F(s_i)$
- (P_1, P_2)-*increasing* for F iff $F(s_i) > F(t_i)$
- (P_1, P_2)-*stable* for F iff $F(t_i) \equiv F(s_i)$.

Example 3.3. Let $|.|$ be the term norm, counting functors as defined in Definition 2.13. Take

$$P_1 = p([a, b, c], f(g(a)), b)$$

and

$$P_2 = p([d, e], g(g(b)), h(b)).$$

Then:

- 1 is (P_1, P_2)-decreasing for $|.|$
- 2 is (P_1, P_2)-stable for $|.|$
- 3 is (P_1, P_2)-increasing for $|.|$.

In the present section, we will always take F equal to $|.|$. We will therefore usually omit the explicit "for $|.|$" addition. Notice that, since $|.|$ maps to the *totally* (strictly) ordered $\mathbb{N}, >$, an argument position is either decreasing or increasing or stable for $|.|$. A similar remark pertains to Definitions 3.5 and 3.6 below. The next definition focuses not on argument positions, but one level higher up the scale, in the restricted context of partition-based measure functions.

Definition 3.5. Let p be a predicate of arity n, and O an associated ordered k-partition. Let P_1 and P_2 be two atoms such that $|P_1|_{p,O} = (v_1, \ldots, v_k)$ and $|P_2|_{p,O} = (w_1, \ldots, w_k)$. Then the ith $(1 \leq i \leq k)$ *component* of O is:

- (P_1, P_2)-*decreasing* iff $v_i > w_i$
- (P_1, P_2)-*increasing* iff $w_i > v_i$
- (P_1, P_2)-*stable* iff $v_i = w_i$.

We will use the notation $O[i]$ to denote the ith component of an ordered partition O.

Example 3.4. Take P_1 and P_2 as in Example 3.3.

- Let $O = (\{1, 2, 3\})$; then its single component $\{1, 2, 3\}$ is (P_1, P_2)-stable.
- For $O' = (\{1, 2\}, \{3\})$, we have:
  — $O'[1] = \{1, 2\}$ is (P_1, P_2)-decreasing
  — $O'[2] = \{3\}$ is (P_1, P_2)-increasing.

Finally, at the level of complete measure functions, we can introduce:

Definition 3.6. Let M be a *mapping* from a set S of atoms to an s-poset $V, >$. Let P_1 and P_2 be two atoms in S. Then M is:

- (P_1, P_2)-*decreasing* iff $M(P_1) > M(P_2)$
- (P_1, P_2)-*increasing* iff $M(P_2) > M(P_1)$
- (P_1, P_2)-*stable* iff $M(P_1) \equiv M(P_2)$.

Of course, Definition 3.6 also will mainly be applied to $|.|_{p,O}$-like measure functions.

Example 3.5. Take P_1, P_2, O, and O' as above.

- $|.|_{p,O}$ is (P_1, P_2)-stable.
- $|.|_{p,O'}$ is (P_1, P_2)-decreasing since $(5,0) \succ_2 (4,1)$.
- If $O'' = (\{3\}, \{1,2\})$, then $|.|_{p,O''}$ is (P_1, P_2)-increasing since $(1,4) \succ_2 (0,5)$.

In the sequel, we will occasionally drop the (P_1, P_2) annotation while using the terminology introduced above (and below) when it is clear which couple of atoms is intended.

The above definitions will be useful in the context of comparing the weight of a goal with the weight of its direct covering ancestor. If we find that the weight increases, we will try to replace the ordered partition in use by one that does result in a decrease. The next few definitions further prepare the way for this operation.

Definition 3.7. Let p be a predicate of arity n, and O an associated ordered k-partition. Let P_1 and P_2 be two atoms with predicate symbol p. Then $O[i]$ is O's *leftmost* (P_1, P_2)-*increasing component* if

1. it is (P_1, P_2)-increasing
2. there is no $1 \leq j < i$ such that $O[j]$ is (P_1, P_2)-increasing

We will occasionally use "leftmost" and "rightmost" in similar contexts without explicitly including a precise definition as the one above.

Definition 3.8. Let p be a predicate of arity n. Let P_1 and P_2 be two atoms with predicate symbol p, and O an associated ordered k-partition. Then we call a component $O[i]$ (P_1, P_2)-*sensitive* if the following two conditions are satisfied:

1. $O[i]$ contains at least one decreasing argument position.
2. If $|.|_{p,O}$ is (P_1, P_2)-increasing and $O[l]$ is O's leftmost increasing component, then $i \leq l$.

Below, we will be interested in replacing a nondecreasing measure by one that does decrease, through a more detailed partitioning of the set of argument positions. In particular, this can be obtained by splitting a sensitive component in, first, a decreasing and, second, an increasing part. Note that, for nondecreasing measures, both parts will be nonempty. However, splitting a component in this way only produces the desired effect if it is not preceded by an increasing component. This is the reason for the second condition above. Finally, note that, while the *leftmost* is therefore the focus of attention among the *increasing* components, below we will be interested in the *rightmost* among the *sensitive* components. Indeed, splitting that one will result in the least drastic, useful weight change (see point (2b) in the proof of Proposition 3.4). The following definition focuses on *nondecreasing* measures.

Definition 3.9. Let p be a predicate of arity n, and O an associated ordered k-partition. Let P_1 and P_2 be two atoms with predicate symbol p such that $|.|_{p,O}$

is not (P_1, P_2)-decreasing. Then we say that $|.|_{p,O}$ has (P_1, P_2)-*potential* iff O has at least one (P_1, P_2)-sensitive component.

Example 3.6. Take P_1, P_2, O, O', and O'' as above.

- $O[1]$ is (P_1, P_2)-sensitive. Since $|.|_{p,O}$ is (P_1, P_2)-stable, this implies that $|.|_{p,O}$ has (P_1, P_2)-potential.
- $O'[1]$ is (P_1, P_2)-sensitive.
- O'' has no (P_1, P_2)-sensitive components.

We have the following property:

Proposition 3.3. Let p be a predicate of arity n, and O an associated ordered k-partition. Let P_1 and P_2 be two atoms with predicate symbol p such that $|.|_{p,O}$ has (P_1, P_2)-potential. Let $O[i]$ be a (P_1, P_2)-sensitive component. Then $\forall 1 \leq j < i : O[j]$ is (P_1, P_2)-stable.

PROOF. Since $|.|_{p,O}$ has (P_1, P_2)-potential, it is either (P_1, P_2)-increasing or (P_1, P_2)-stable. In the latter case, the result is immediate. In the former case, it follows from the second condition in Definition 3.8. □

We now formally introduce a refinement operation for ordered partitions and partition-based measures.

Definition 3.10. Let $O = (C_1, \ldots, C_k)$ be an ordered k-partition of some set V. Then the ordered $k+1$-partition $O' = (C'_1, \ldots, C'_{k+1})$ is called an *i-refinement* of O $(1 \leq i \leq k)$ if:

- $C_i = C'_i \cup C'_{i+1}$
- $\forall 1 \leq j < i : C_j = C'_j$
- $\forall i < j \leq k : C_j = C'_{j+1}$.

Example 3.7. When defined as above, both O' and O'' are 1-refinements of O.

As in the following definition, we will occasionally use the term "refinement" in contexts where the actual i index does not matter.

Definition 3.11. Let p be a predicate of arity n, and O an associated ordered partition. Let O' be a refinement of O. Then we call $|.|_{p,O'}$ a *refinement* of $|.|_{p,O}$.

Finally, we are in a position to introduce the following key concept:

Definition 3.12. Let p be a predicate of arity n, and $O = (C_1, \ldots, C_k)$ an associated ordered k-partition. Let P_1 and P_2 be two atoms with predicate symbol p such that $|.|_{p,O}$ has (P_1, P_2)-potential. Let C_l be O's rightmost (P_1, P_2)-sensitive component. Then $|.|_{p,O'}$ is the *tight (P_1, P_2)-decreasing refinement* of $|.|_{p,O}$ if O' is an l-refinement of O and:

- $O'[l] = \{i \in C_l \mid i$ is (P_1, P_2)-decreasing or (P_1, P_2)-stable$\}$
- $O'[l+1] = \{i \in C_l \mid i$ is (P_1, P_2)-increasing$\}$.

Example 3.8. In our running example, $|.|_{p,O'}$ is the tight (P_1, P_2)-decreasing refinement of $|.|_{p,O}$. Notice that no refinement of $|.|_{p,O''}$ is (P_1, P_2)-decreasing.

Definition 3.12 is central in the automatic search for good measure functions. Indeed, when a given measure function does not result in a weight decrease, we will (try to) replace it by its tight decreasing refinement, thus enabling further unfolding. Before we present the details of the procedure in the algorithm below, we first establish that tight decreasing refinements are well defined.

Proposition 3.4. Let p be a predicate of arity n, and $O = (C_1, \ldots, C_k)$ an associated ordered k-partition. Let P_1 and P_2 be two atoms with predicate symbol p, such that $|.|_{p,O}$ has (P_1, P_2)-potential. Then $|.|_{p,O'}$, the tight (P_1, P_2)-decreasing refinement of $|.|_{p,O}$, exhibits the following properties:

1. *$|.|_{p,O'}$ is (P_1, P_2)-decreasing.*
2. *Any other (P_1, P_2)-decreasing refinement, $|.|_{p,O''}$, of $|.|_{p,O}$, obtained by splitting one of O's components in two subsets, one of which contains all of its increasing argument positions, has the following property:*
   There is no atom P with predicate symbol p such that $|P|_{p,O''} \succ_{k+1} |P|_{p,O'}$.

PROOF. Definition 3.9 assures that O has a rightmost (P_1, P_2)-sensitive component. Let l be its index. Then we can argue as follows:

1. First we note that $O'[l]$ is (P_1, P_2)-decreasing. Furthermore, Proposition 3.3 implies that all O' components to the left of $O'[l]$ are (P_1, P_2)-stable. It follows that $|.|_{p,O'}$ is (P_1, P_2)-decreasing.
2. The desired property follows from the following considerations:
   (a) Splitting a component which is *not* (P_1, P_2)-sensitive, if possible, does not result in a (P_1, P_2)-decreasing measure.
   (b) Suppose we split a (P_1, P_2)-sensitive component $O[i]$ with $i \neq l$. It follows that $i < l$. Suppose that $|P|_{p,O} = (v_1, \ldots, v_k)$.
   Let

   $$|P|_{p,O'} = (v'_1, \ldots, v'_i, \ldots, v'_l, v'_{l+1}, \ldots, v'_{k+1})$$

   and

   $$|P|_{p,O''} = (v''_1, \ldots, v''_i, v''_{i+1}, \ldots, v''_l, \ldots, v''_{k+1}).$$

   Then $\forall 1 \leq j < i : v_j = v'_j = v''_j$. Moreover, $v'_i = v_i$ and $v_i \geq v''_i$.
   - If $v_i > v''_i$, then obviously $|P|_{p,O'} \succ_{k+1} |P|_{p,O''}$.
   - If $v_i = v''_i$, then $v''_{i+1} = 0$. Moreover, $\forall i+1 \leq j < l : v'_j = 0 \Rightarrow v''_{j+1} = 0$. Finally, $v'_l = 0 \Rightarrow v'_{l+1} = v_l = v''_{l+1}$ while, of course, $\forall l+1 < j \leq k+1 : v'_j = v_{j-1} = v''_j$. So, either $v'_j \neq 0$ for some $i < j \leq l$, from which it follows that $|P|_{p,O'} \succ_{k+1} |P|_{p,O''}$, or $|P|_{p,O'} = |P|_{p,O''}$.

   (c) The property is immediate for any refinement where some nonincreasing argument positions are also put into $O''[l+1]$. □

The second property above is the motivation for including "tight" in the naming done in Definition 3.12. It ensures that we are relatively "conservative" when taking refinements. Bigger weights are generally better than smaller because they allow a longer decrease, which means more unfolding potential. It would, of course, be possible to further "tighten" Definition 3.12 by splitting off sufficiently large *subsets* of the set of increasing argument positions in the rightmost increasing component. We have decided not to do this because it would complicate the

method both conceptually and computationally. Moreover, we conjecture that there would almost never be a substantial gain in unfolding capacity. A detailed investigation into the validity of the latter claim is outside the scope of this paper.

3.3.2. The Algorithm. Above, we developed partition-based measure functions, thus generalizing the subset-based functions introduced in Definition 2.14. We now use them as a basic ingredient of an advanced automatic algorithm for sensible finite unfolding.

In Algorithm 3.1 below, we choose $R_0, R_1, \ldots, R_N$ as indicated in Section 2.3. This means that a goal (G', j) such that $(G', j) >_{pr} (G, i)$ *covers* (G, i) if their *selected atoms contain the same recursive predicate symbol.* For $F_1, \ldots, F_N$, we take $|.|_{p,O_p}$-like functions, one per recursive predicate, where, as before, we associate with a *goal* the value of the relevant measure function on its *selected atom.* The optimal partitions O_p to be used for each recursive predicate p and the computation rule R are dynamically fixed while executing the algorithm.

Algorithm 3.1

Input
a definite program P
a definite goal $\leftarrow A$
Output
a finite SLD-tree τ *for* $P \cup \{\leftarrow A\}$
Initialization
$\tau := \{(\leftarrow A, 1)\}$ {* *an SLD-tree with a single derivation* *}
$Pr := \emptyset$ {* *in* Pr, *the* $>_{pr}$*-relation will be constructed* *}
Terminated $:= \emptyset$
Failed $:= \emptyset$
For each recursive predicate p/n *in* P: $O_p := (\{1, \ldots, n\})$
{* *We set out with 1-partitions*
grouping all argument positions in a single component *}
While *there exists a derivation* D *in* τ *such that* $D \notin$ *Terminated* **do**
Let (G, i) *name the leaf of* D
If $(G, i) = (\square, i)$
Then {* (G, i) *is a success node* *}
add D *to Terminated*
Else
{* *First, we try to determine* $R(G, i)$ *}
Select the leftmost atom $p(t_1, \ldots, t_n)$ *in* G *such that*
one of the following (mutually exclusive) conditions is satisfied:
- (G, i) *has no direct covering ancestor*
- (G', j) *is the direct covering ancestor of* (G, i) *and*
  $|.|_{p,O_p}$ *is* $(R(G', j), p(t_1, \ldots, t_n))$*-decreasing*
- (G', j) *is the direct covering ancestor of* (G, i) *and*
  $|.|_{p,O_p}$ *has* $(R(G', j), p(t_1, \ldots, t_n))$*-potential and*
  $|.|_{p,O'_p}$ *is its tight* $(R(G', j), p(t_1, \ldots, t_n))$*-decreasing refinement and*
  τ *remains subset-wise founded with respect to*
  $((R_0, R_1, \ldots, R_N), (|.|_{p_1,O_{p_1}}, \ldots, |.|_{p,O'_p}, \ldots, |.|_{p_N,O_{p_N}}))$ (*)

If *such an atom* $p(t_1, \ldots, t_n)$ *cannot be found*
Then {* (G, i) *becomes a dangling leaf* *}
Add D *to Terminated*
Else
$R(G, i) := p(t_1, \ldots, t_n)$
If $R(G, i)$ *was selected on the basis of the third condition above*
Then {* *Register the new partition* *}
$O_p := O'_p$
Let Resolvents(G, i) *name the set of all derivation steps that can be performed*
If *Resolvents*$(G, i) = \emptyset$
Then {* (G, i) *is a failure node* *}
Add D *to Terminated and Failed*
Else
{* *Extend the derivation* *}
Expand D *in* τ *with the elements of Resolvents*(G, i)
Let Descend$(R(G, i), i)$ *name the set of all pairs* $((R(G, i), i), (B\theta, j))$, *where*
— B *is an atom in the body of a clause applied in an element of Resolvents*(G, i)
— θ *is the corresponding mgu*
— j *is the number of the corresponding descendant of* (G, i)
Apply θ *to the affected elements of* Pr
Add the elements of Descend$(R(G, i), i)$ *to* Pr
Endwhile

Example 3.9. For the program and query in Example 3.2, Algorithm 3.1 produces exactly the SLD-tree depicted in Figure 1. Indeed, the first unfolding is possible on the basis of the "no covering ancestor" condition. When we try to continue, we notice that $|.|_{p\_c,(\{1,2\})}$ is not decreasing. But it has potential, and its tight decreasing refinement, $|.|_{p\_c,(\{1\},\{2\})}$, is the actual measure function used in Example 3.2.

Theorem 3.1. Algorithm 3.1 terminates. The resulting SLD-tree τ *is finite.*

PROOF. We first note that, as long as changes in the measure functions do not occur, the computation rule R is developed in a way that renders Algorithm 3.1 an instance of Algorithm 2.1. The result then follows from Theorem 2.4. It remains to be shown that the operation of replacing a measure function by its tight decreasing refinement occurs only a finite amount of times.

First note that there is only a finite number of distinct measure functions considered at any moment, one for each recursive predicate in the input progam P. Next, the operation of taking a tight decreasing refinement involves splitting a nonempty set of argument positions in two disjunct nonempty sets of argument positions. The result follows from the fact that any predicate symbol has only a finite number of argument positions. □

Concluding, we have proposed, formalized, and proven correct (terminating) an automatic unfolding method for definite logic programs. This method generalizes our previous approach where increasing argument positions were *deleted* from the considered set. Instead of employing this straightforward, but drastic, technique,

we now *shift* such argument positions to the right in the considered partition, where they determine a lexicographically less important component of the resulting weight. Unlike before, doing so allows *future consideration* of the corresponding arguments. This is conceptually more elegant, and might create extra unfolding opportunities, as Example 3.2 illustrates. We conjecture that partition-based measures are strictly more powerful (i.e., other things equal, produce at least equally large SLD-trees) than those based on sets. (But see Section 6.3.) Intuitively, this seems obvious. Indeed, set-based unfolding is equivalent to considering only the leftmost component in partition-based unfolding. We do not include a formal analysis of this issue, but instead proceed to exhibit an important simplification of the above algorithm.

3.4. Relaxing Monotonicity

We will not include a complete formal complexity analysis of Algorithm 3.1. However, one important observation should be made: it is possible to implement such weight-based unfolding algorithms in such a way that their behavior is *linear in the size of the generated SLD-tree*, as long as no change in measure function is required. Indeed, as shown below, efficient labeling techniques ensure that for any atom, deciding whether it should be compared with the selected atom of an ancestor goal, and if so, carrying out the comparison, can be done without any search.

However, the requirement that, upon refining a measure function, the whole SLD-tree generated thus far should be checked to verify whether it remains subsetwise founded destroys this linearity property. We already pointed out in [37] that, in practice, this rechecking can safely be ignored without damaging termination. Experiments in [22] confirmed this conjecture, but no formal proof was given. In this subsection, we present a brief formal development of this issue.

First, we generalize the notion of a well-founded measure.

Definition 3.13. Let $V, >_V$ be an s-poset. A *nearly-founded measure*, f, *on* $V, >_V$ is a function from $V, >_V$ to some well-founded set $W, >_W$ such that the following holds for only a *finite* number of pairs of elements v_i and $v_j \in V$:

$$v_i >_V v_j \text{ and not } (f(v_i) >_W f(v_j)).$$

Unlike a well-founded measure (Definition 3.5), a nearly founded measure does not have to be monotonic. But it is "almost" monotonic: there are only a finite number of offending pairs in the mapped set. We now introduce the notion of a subsetwise *nearly* founded SLD-tree, and show that it still guarantees finiteness.

Definition 3.14. An SLD-tree τ is *subsetwise nearly founded* if:

1. There exists a finite number of sets, $C_0, \ldots, C_N$, such that $G_\tau = \cup\{C_i \mid i \leq N\}$.
2. For each $i = 1, \ldots, N$, there exists a nearly founded measure $f_i: C_i, >_\tau \rightarrow W_i, >_i$.
3. For each $(G, k) \in C_0$ and each derivation D in τ containing (G, k):
   - D is finite or
   - there exists a descendant (G', j) of (G, k) in D such that $(G', j) \in C_i$ for some $i > 0$.

Just as for subsetwise foundedness (see Section 2.2), we will occasionally refer to an SLD-tree being subsetwise nearly founded with respect to a pair $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$.

Theorem 3.2. An SLD-tree τ is finite iff it is subsetwise nearly founded.

The proof is a straightforward adaptation of the proof for [8, Theorem 2.2] (or Theorem 2.2 above). We include it for completeness.

PROOF.

- If τ is finite, then it is subsetwise founded, and therefore subsetwise nearly founded.
- Conversely, suppose that τ is subsetwise nearly founded and infinite. Then it contains an infinite derivation D. From the first condition in Definition 3.14, it follows that there must be a C_i such that $C_i \cap D$ is infinite. In other words, there is some C_i containing an infinite sequence $(G_0, i_0) >_\tau (G_1, i_1) >_\tau \cdots$.
  — Suppose $i > 0$. Then $f_i((G_{j_1}, i_{j_1})) >_i f_i((G_{j_2}, i_{j_2})) >_i \cdots$ is an infinite sequence in $W_i, >_i$, contradicting the well-foundedness of $W_i, >_i$.
  — This leaves $i = 0$ as the only possibility. But then condition 3 of Definition 3.14 implies that $D \cap \cup\{C_i \mid i > 0\}$ is infinite, which again requires the existence of some $C_i, i > 0$ such that $C_i \cap D$ is infinite. □

We modify Algorithm 3.1.

Algorithm 3.2. Algorithm 3.1 remains almost completely unchanged. We just delete the fourth conjunct, marked (*), from the third condition enabling an atom selection. We do not reproduce the whole remaining algorithm description here.

Proposition 3.5. Algorithm 3.2 constructs a subsetwise nearly founded SLD-tree.

PROOF. Most of the reasoning is identical to what has been presented in the subsetwise founded case. We only point out that measure functions are indeed nearly founded since, when refining a partition, the resulting measure function:

1. might be nonmonotonic on the *finite* subtree constructed thus far
2. will be (subsetwise) strictly decreasing on newly added nodes.

Furthermore, as argued in the proof for Theorem 3.1, replacing a measure function by its tight decreasing refinement occurs only a finite number of times. Together, these considerations imply that (subsetwise) nonmonotonicity will hold for only a finite amount of pairs in the overall tree. □

This implies:

Theorem 3.3. Algorithm 3.2 terminates. The resulting SLD-tree τ is finite.

Having established its termination, we briefly return to the performance properties of Algorithm 3.2. It is important to realize that checking the unfoldability of a certain atom can be done *without searching branches* in the SLD-tree, thus giving rise to the above-mentioned linearity property. Indeed, there is just *a single test* involved: a weight comparison with the selected atom of the direct covering

ancestor. Finding out whether such a direct covering ancestor exists, and if so, spotting it (in other words, maintaining and using the $>_{pr}$-relation) can be efficiently implemented through the use of lists of relevant node numbers. With an atom in a goal, one such list is associated. It registers per atom for every recursive predicate the closest $>_{pr}$-ancestor node where the selected atom contains that predicate symbol.

Example 3.10. Consider the following schematic program with three recursive predicates:

$p \leftarrow q, p$

$q \leftarrow r$

$r \leftarrow p.$

An annotated SLD-derivation for $\leftarrow p$ is depicted in Figure 2.

Since the program contains three recursive predicates, the length of the lists associated to the atoms in goals is 3. In each list, the first position corresponds to p, the second to q, the third to r. Elements of the list associated to an atom A are the indices of the most recent goal node where a $>_{pr}$-ancestor atom of A with the corresponding predicate symbol was selected. A few concrete examples will clarify this somewhat complex description:

- In (2), the two descendant atoms of the selected p-atom in (1) both get a 1 on the first list position. The other list positions, of course, remain "vacant."
- In (4), the left p-atom has $>_{pr}$-ancestors of every kind; the right one does not.
- In (5), the q-atom and the left p-atom both descend from the p-atom selected in (4). The right p-atom, however, does not.

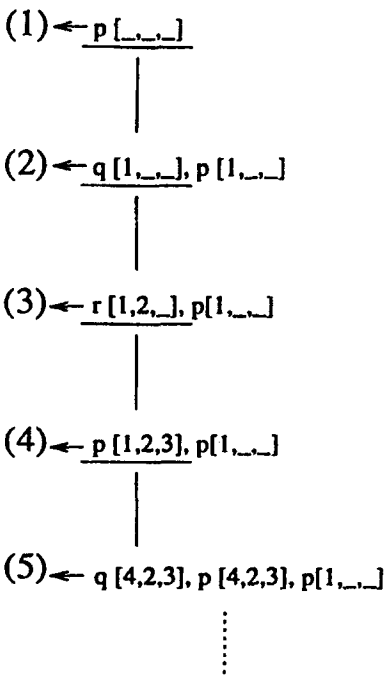

FIGURE 2. Direct covering ancestor annotation.

In this way, direct covering ancestors can immediately be spotted. Consider goal (5) above:

- Selecting the q-atom requires a weight comparison with the selected atom of (2).
- The left p-atom must be compared with the selected one in (4).
- Finally, the right p-atom, not descending from (4)'s selected atom, can be unfolded if its weight is less than the one associated to the original p-goal, selected in (1).

It is clear that the above results do not depend on the use of partition-based measures; they carry over straightforwardly to methods using set-based measures. Some further implementational details in the latter context can be found in [22].

At this point, it is interesting to include a brief comparison with some other criteria to control unfolding as they have been proposed in the literature. First, four possible tests are mentioned in [2]. Unfolding is prohibited when the considered atom is a variant of, is an instance of, is more general than, has a common instance with an atom selected in an ancestor goal. Particularly, the "instance of" criterion has also been used, in e.g., [16] and [29]. It is well known that neither of the four criteria is "safe," in the sense that termination of unfolding is not guaranteed. Or, using the terminology introduced in [4], they are not complete for the class of all definite logic programs. (See also [3].) However, another drawback, more immediately relevant for the discussion at hand, can be observed. Indeed, any such criterion unavoidably necessitates searching through the ancestor goals, thus destroying the above-mentioned linearity property. Next, several criteria of increasing sophistication are proposed in [45]. The simplest one just involves counting the number of occurrences of the various predicate symbols in selected atoms. This can obviously be implemented such that linear behavior results. The same holds for some (but not all) of the more sophisticated methods where argument weights in selected atoms are considered. For tests that rely on structure comparisons, searching the goal stack seems inevitable. Section 7 contains a further discussion of the loop prevention methods described in [45]. Finally, [40] proposes a method for partial deduction which involves not only unfoldings, but also the introduction of new predicates and foldings. A detailed discussion leads too far, but it can be noted that the method behaves linearly with respect to a parameter related to the size of the SLD-tree upon unfolding. Our analysis shows that similar results can be obtained through the use of weight-based unfolding.

Theoretically, Algorithm 3.2 might build larger SLD-trees than Algorithm 3.1. Indeed, its third condition for atom unfoldability is more easily satisfied since it does not contain (*). However, switching to the tight decreasing refinement of the measure under consideration, and thus extending the given derivation, might be at the cost of diminished unfolding capabilities in other derivations. The latter possibility would be excluded through the use of separate partitions per chain of covering nodes. In practice, it turns out that all such technicalities are of very minor importance: arguments of most programs behave in a way sufficiently regular to mask these details. (Condition (*), e.g., is usually satisfied when taking a tight decreasing refinement.) We therefore simply note that we have eliminated a possible source of considerable inefficiency from Algorithm 3.1, while still guaranteeing termination. And we will do likewise for all weight-based unfolding algorithms to be presented below.

4. CONSIDERING THE CONTEXT

4.1. Introduction

Above, we introduced partitions of the set of and priorities among argument positions of *a goal's selected atom.* We illustrated how this generalization of our earlier work brings extra unfolding power. In the present section, we will take yet another step towards increased power.

Indeed, we will return to our original objective, mentioned in the introduction to Section 3: taking into account arguments of several/all atoms in the goal, not only the candidate for selection. In other words, the basic idea underlying our approach is kept intact: the weight of successive nodes with the same selected atom in a chain of covering goals should decrease. But weights will no longer be assigned solely on the basis of the selected atom. A recasting of Example 3.2 shows what we have in mind.

Example 4.1. Consider the following program:

$produce([], []) \leftarrow$

$produce([X \mid Xs], [X \mid Ys]) \leftarrow produce(Xs, Ys)$

$consume([]) \leftarrow$

$consume([X \mid Xs]) \leftarrow consume(Xs)$

and query:

$\leftarrow produce([1, 2 \mid X], Y), consume(Y)$

We apply Algorithm 2.1 *imposing a coroutining computation rule* and choosing the pair $((R_0, R_1, R_2), (F_1, F_2))$ as follows:

- $R_0 = \emptyset$
- $R_1 = \{R(G, i)$ containing *produce*$\}$
- $R_2 = \{R(G, i)$ containing *consume*$\}$
- $F_1 = |.|_{produce,(\{1,2\})}$
- $F_2 = (|.|_{produce,\{1,2\}}, |.|_{consume,\{1\}})$.

The resulting SLD-tree is depicted in Figure 3. Selected atoms are underlined, and nodes are annotated with their weight according to F_1 or F_2. It can be noted that the second *consume*-unfolding is not allowed using a measure function just taking the *consume*-argument into account.

In this extended context, a major difficulty is the dynamic nature of atom occurrences in goals. This is illustrated by the left branch of the SLD-tree in Figure 3. As long as the goals to be measured basically look the same, containing one *produce* and one *consume* atom, a measure function like F_2 indeed makes sense. But this is not the case for the last nonempty goal in the above left branch, where the *produce* call has "disappeared" (indicated by a "-" in the associated weight couple). Worse even, in general, more than one atom with the same predicate symbol might appear in one goal, and it is by no means immediately clear what should be compared with what in such circumstances.

Our work in this section will be presented as follows. First, we exhibit in detail a rather straightforward solution to the problems indicated above, and show that the

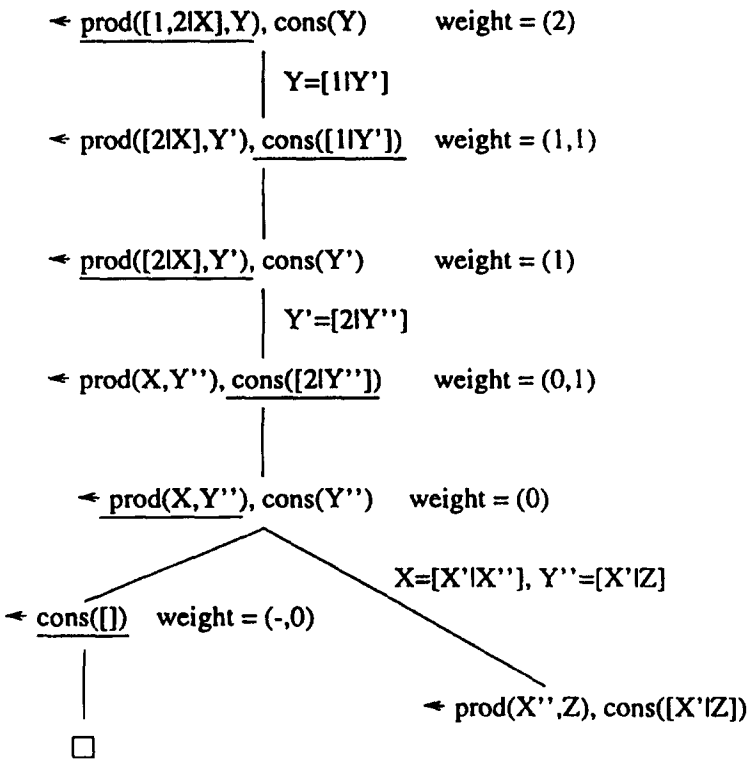

FIGURE 3. Unfolding when considering other atoms.

resulting algorithms possess sufficient power to handle typical producer–consumer coroutining applications. Second, we demonstrate how a more satisfactory treatment of an annoying issue in "standard" unfolding is also enabled. Finally, we briefly describe a more sophisticated approach.

4.2. Handling Coroutining

In this subsection, we present two algorithms suitable for unfolding under coroutining-like computation rules. The first one, Algorithm 4.1, is simply a properly specialized version of Algorithm 2.1. Algorithm 4.2, on the other hand, is a semi-automatic generalization of Algorithm 3.2. Along the way, we need to generalize a number of concepts introduced above.

4.2.1. Tuning the Basic Algorithm. First, we want to allow argument positions of nonselected atoms among those determining a goal's weight. On the other hand, we will stick to comparing a goal with its direct covering ancestor. So, the selected atom will continue to play a major role, as actually seems quite natural in the context of unfolding logic programs. (See also [3].) We set out with the following definition.

Definition 4.1. Let P be a program containing a (recursive) predicate symbol p. A *context considering ordered k-partition* (*cco-k-partition*) *associated to p in P* is a k-tuple $O = (\{i_{11}, \ldots, i_{1j}\}, \ldots, \{i_{k1}, \ldots, i_{kl}\})$ satisfying the following conditions:

1. O has *two kinds of components.* Some, which we will call *p-components*, consist of argument positions of p. The others contain argument positions of recursive predicate symbols in P, and will occasionally be named *non-p-components.*
2. The p-components together form an *ordered partition of p's set of argument positions.*
3. Argument positions of recursive atoms in P (including p) can appear in at most one non-p-component.

Example 4.2. For the program in Example 4.1, the following are some cco-2-partitions associated to *consume*:

- $(\{1_{produce}, 2_{produce}\}, \{1\})$
- $(\{1_{produce}\}, \{1\})$
- $(\{1\}, \{2_{produce}, 1_{consume}\})$
- $(\{1\}, \{1_{consume}\})$.

Some further examples, this time of cco-partitions associated to *produce*, are:

- $(\{1, 2\})$
- $(\{1\}, \{1_{consume}\}, \{2\})$

while the next two tuples are *not* legitimate *produce* associated cco-partitions:

- $(\{1\}, \{1_{consume}\})$
- $(\{1_{consume}\}, \{1, 2\}, \{1_{consume}\})$.

Note that Definition 4.1 is an extension of the notion "associated ordered k-partition" introduced above. Indeed, we expand the earlier partition of a predicate's set of argument positions with "components" containing some argument positions of other relevant predicates. Although possible, we have decided against requiring the presence of *all* argument positions of *all* (recursive) predicates since this would often introduce a great amount of irrelevant information in a goal's measure function. The same consideration is the motivation for only considering *recursive* predicate symbols. Finally, note that we anticipate a distinction between p in the selected atom and the same predicate symbol occurring in nonselected atoms: argument positions of p appearing in non-p-components refer to such occurrences.

Before we can actually introduce the generalized measure functions, we need to provide for "absent" arguments.

Definition 4.2. We define $\mathbb{N}_b = \mathbb{N} \cup \{\bot\}$. We extend the usual order relation, $>$, on $\mathbb{N}$ with $0 > \bot$. We extend the usual addition, $+$, on $\mathbb{N}$ with $x + \bot = \bot + x = \bot$.

$\bot$ is an extra "bottom" element. It will serve as the image of a measure function on an absent argument. Notice that we have overloaded the $>$, $+$, and $=$ symbols, and that the result of an addition involving $\bot$ is $\bot$. Of course, $\mathbb{N}_b, >$ is a well-founded set, and so is $\mathbb{N}_b^k, \succ_k$ for any $k > 1$.

We need an operation on finite subsets of $\mathbb{N}_b$, delivering their maximum element.

Definition 4.3. We define $max: \mathcal{P}(\mathbb{N}_b) \to \mathbb{N}_b$ as follows:

- $max(\emptyset) = \perp$
- $max(\{v_1, \ldots, v_n\}) = v_i$ such that $\forall 1 \leq j \leq n : j \neq i \Rightarrow v_i > v_j$.

Note that, since $>$ is a *total* order on $\mathbb{N}_b$, max is well defined on finite sets and its result uniquely determined. We continue:

Definition 4.4. Let G be a goal consisting of a number of atoms, one of which has been selected for unfolding, p an n-ary predicate symbol, and $1 \leq i \leq n$. Then we define

$$M(G,p,i) = max(\{|t_i| \mid t_i \text{ is the term occurring as } i\text{th argument of some } \textit{nonselected} \text{ atom } p(t_1, \ldots, t_n) \text{ in } G\}).$$

Finally, we can formulate the following generalization of Definition 4.2:

Definition 4.5. Let P be a program and p an n-ary (recursive) predicate symbol appearing in P. Let O be a cco-k-partition associated to p in P. Then we define $\|.\|_{p,O}: \{G \mid G \text{ is a goal in the language underlying } P \text{ whose selected atom has predicate symbol } p\} \to \mathbb{N}_b^k, G \mapsto (v_1, \ldots, v_k)$ as follows:

- If $O[r] = \{i_{r1}, \ldots, i_{rs}\}$ is a *p-component* and $p(t_1, \ldots, t_n)$ is G's selected atom, then $v_r = |t_{i_{r1}}| + \cdots + |t_{i_{rs}}|$.
- If $O[r] = \{i_{r1,p_{r1}}, \ldots, i_{rj,p_{rj}}\}$ where the p_{rl}-subscripts denote recursive predicate symbols in P, then $v_r = M(G, p_{r1}, i_{r1}) + \cdots + M(G, p_{rj}, i_{rj})$.

Example 4.3. The weight annotations in Figure 3 correspond to:

- $O_{produce} = (\{1, 2\})$
- $O_{consume} = (\{1_{produce}\}, \{1\})$ (taking $O_{consume}[1] = \{1_{produce}, 2_{produce}\}$ gives the same results).

Example 4.4. Suppose

- p, q and r are recursive predicates
- $G = \leftarrow q(f(a)), p(f(f(a)), f(a)), p(f(a), a), q(f(f(a)))$
- in G, the first p-atom is selected
- $O = (\{1_q, 2_p\}, \{1, 2\}, \{1_p, 1_r\})$.

Then $\|G\|_{p,O} = (2+0, 2+1, 1+\perp) = (2, 3, \perp)$.

We have the following equivalent of Proposition 3.2:

Proposition 4.1. Let P be a program, and p a predicate symbol appearing in P. Let O be a cco-k-partition associated to p in P. Let τ be an SLD-tree for P and S_τ a subset of G_τ such that all goals in S_τ have a selected atom with predicate p. Then $\|.\|_{p,O}$ is a well-founded measure on $S_\tau, >_\tau$ iff it is monotonic.

PROOF. The proposition follows immediately from Definitions 2.5 and 4.5 and the well-foundedness of $I\!N_b^k, \succ_k$. □

Suppose now that a (coroutining) computation rule R is fixed, to be used while unfolding a goal with respect to a program P. Moreover, also given are cco-partitions O_p, one for every recursive predicate p in P. Then we can introduce the following specialized version of Algorithm 2.1:

Algorithm 4.1. As before, we associate one R_k to every recursive predicate. If p_k is a recursive predicate with associated class R_k, then $F_k = \|.\|_{p_k,O_{p_k}}$. Adapting the code of the algorithm is straightforward; we do not reproduce it here.

Theorem 4.1. Algorithm 4.1 terminates. The resulting SLD-tree τ is finite.

PROOF. $((R_0, R_1, \ldots), (F_1, \ldots))$ is chosen such that it indeed determines a hierarchical prefounding of the complete SLD-tree τ_0, resulting for the program P and a given goal under the computation rule R. The result follows immediately from Theorem 2.4. □

Example 4.5. We have formalized the intuitions underlying Example 4.1. Indeed, instead of the ad hoc couple of "measure" functions used for F_2 there, we can now take $F_2 = \|.\|_{consume,(\{1_{produce}\},\{1\})}$ (and $F_1 = \|.\|_{produce,(\{1,2\})}$). And, also, the disappearing *produce* call can now be dealt with properly within our framework. An application of Algorithm 4.1 using these ingredients reproduces the SLD-tree depicted in Figure 3. Goal weights result as indicated, except $(\_, 0)$ which becomes $(\bot, 0)$.

Example 4.6. As a concrete example of coroutining behavior, we consider the well-known permutation sort program:

(1) $sort(X, Y) \leftarrow perm(X, Y), ord(Y)$
(2) $perm([], []) \leftarrow$
(3) $perm([X \mid Xs], [Y \mid Ys]) \leftarrow del(Y, [X \mid Xs], Z), perm(Z, Ys)$
(4) $del(X, [X \mid Xs], Xs) \leftarrow$
(5) $del(X, [Y \mid Ys], [Y \mid Z]) \leftarrow del(X, Ys, Z)$
(6) $ord([]) \leftarrow$
(7) $ord([X]) \leftarrow$
(8) $ord([X, Y \mid Z]) \leftarrow X \leq Y, ord([Y \mid Z])$

We do not include explicit clauses for $\leq$, and simply assume that an $X \leq Y$ call can be evaluated when both arguments are ground. We want to build an SLD-tree for the query

$\leftarrow sort([5, 2 \mid X], Y)$

using the following coroutining computation rule (the actions are listed according to priority; "possible" refers to associated weight behavior):

1. Evaluate $X \leq Y$ if both arguments are ground.
2. Unfold a *del* call if possible.
3. Unfold an *ord* call if possible, provided its argument is not an uninstantiated variable.
4. Unfold a *perm* call if possible.

We will use the following R_i classes:

- $R_0 = \{R(G,i)$ containing $\leq\} \cup \{R(G,i)$ containing $sort\}$
- $R_1 = \{R(G,i)$ containing $del\}$
- $R_2 = \{R(G,i)$ containing $ord\}$
- $R_3 = \{R(G,i)$ containing $perm\}$

and as associated cco-partitions:

- $O_{del} = (\{1,2,3\})$
- $O_{ord} = (\{1_{perm}\},\{1\})$
- $O_{perm} = (\{1\},\{2\})$.

Rather than reproducing the complete SLD-tree resulting from an application of Algorithm 4.1, we select one SLD-derivation and depict it in Figure 4. Each node is annotated with a label showing its identifier $(a,b,\ldots)$, the one of its direct covering ancestor, and its weight under the relevant measure function. Links are enhanced with a number indicating the clause used in that particular unfolding.

Especially noteworthy is the unfolding carried out between nodes (i) and (j). It would be prohibited when using a measure function based solely on the *ord* atom itself.

4.2.2. Automatically Refining Measure Functions. Having established that Algorithm 4.1 can cope with coroutining when apt cco-partitions are provided, we want to take a further step. Indeed, it seems appropriate to free the user from the burden of choosing the right partitions. In other words, we want to obtain an algorithm similar to Algorithm 3.2 (or Algorithm 3.1), with the difference that some computation rule preferences are specified by the user, as was done in Example 4.6 above.

First, we introduce proper generalizations of the definitions in Section 3.3.1.

Definition 4.6. Let P be a program containing an n-ary predicate symbol p. Let O be a cco-k-partition associated to p in P. Let G_1 and G_2 be two goals in the language underlying P. Let the selected atom of G_1 and G_2 be $p(t_1,\ldots,t_n)$ and $p(s_1,\ldots,s_n)$, respectively. Finally, let $\|G_1\|_{p,O} = (v_1,\ldots,v_k)$ and $\|G_2\|_{p,O} = (w_1,\ldots,w_k)$. Then we define the following:

- An *argument position i in a p-component* of O is
  - (G_1,G_2)-*decreasing* iff $|t_i| > |s_i|$
  - (G_1,G_2)-*increasing* iff $|s_i| > |t_i|$
  - (G_1,G_2)-*stable* iff $|t_i| = |s_i|$.
- An *argument position i_q* is
  - (G_1,G_2)-*context-decreasing* iff $M(G_1,q,i) > M(G_2,q,i)$
  - (G_1,G_2)-*context-increasing* iff $M(G_2,q,i) > M(G_1,q,i)$
  - (G_1,G_2)-*context-stable* iff $M(G_1,q,i) = M(G_2,q,i)$.

  When we consider arguments as elements of a non-p-component of a cco-partition, we will occasionally describe their behavior without the explicit "context" addition. In such cases, it is clear that we refer to M- and not $|.|$-values.
- The *ith component* of O is
  - (G_1,G_2)-*decreasing* iff $v_i > w_i$

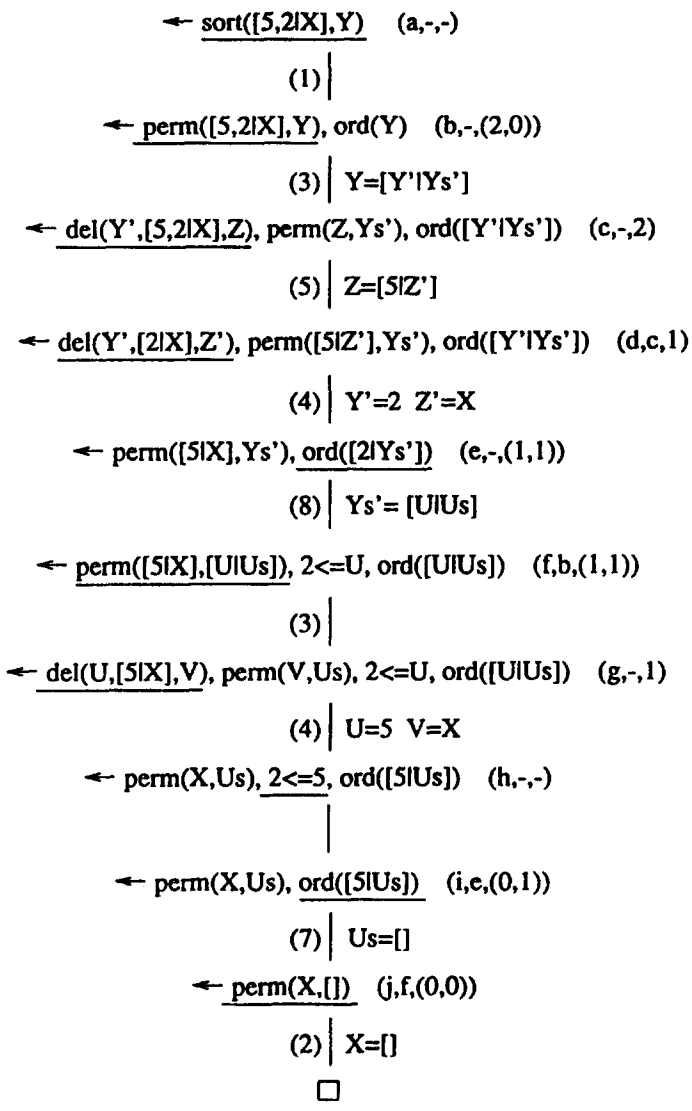

FIGURE 4. A coroutining SLD-derivation.

— (G_1, G_2)-*increasing* iff $w_i > v_i$
— (G_1, G_2)-*stable* iff $v_i = w_i$.

- $\|.\|_{p,O}$ is
  — (G_1, G_2)-*decreasing* iff $\|G_1\|_{p,O} \succ_k \|G_2\|_{p,O}$
  — (G_1, G_2)-*increasing* iff $\|G_2\|_{p,O} \succ_k \|G_1\|_{p,O}$
  — (G_1, G_2)-*stable* iff $\|G_1\|_{p,O} = \|G_2\|_{p,O}$.
- $O[i]$ is O's *leftmost* (G_1, G_2)-*increasing component* if
  1. it is (G_1, G_2)-increasing
  2. there is no $1 \leq j < i$ such that $O[j]$ is (G_1, G_2)-increasing.

- We call a component $O[i]$ (G_1, G_2)-*sensitive* if the following two conditions are satisfied:
  1. $O[i]$ contains at least one (G_1, G_2)-decreasing argument position.
  2. If $\|.\|_{p,O}$ is (G_1, G_2)-increasing and $O[l]$ is O's leftmost increasing component, then $i \leq l$.

Definition 4.7. Let P be a program containing an n-ary predicate symbol p. Let O be a cco-k-partition associated to p in P. Let G_1 and G_2 be two goals in the language underlying P whose selected atom contains p. Suppose $\|.\|_{p,O}$ is *not* (G_1, G_2)-decreasing. Then we say that $\|.\|_{p,O}$ has

- *internal* (G_1, G_2)-*potential* iff O has at least one (G_1, G_2)-sensitive component
- *external* (G_1, G_2)-*potential* iff there is at least one argument position i_q of some recursive predicate q in P for which the following two conditions hold:
  1. i_q is (G_1, G_2)-context-decreasing
  2. for every non-p-component $O[j]$ of O: $i_q \notin O[j]$ (we will call i_q *context-absent in* O).

We will occasionally say that a measure function $\|.\|_{p,O}$ has (G_1, G_2)-*potential* if either of the above two conditions is satisfied.

Just like before, we are interested in refining obsolete (i.e., nondecreasing) measure functions. The notion of *internal* potential with respect to a pair of goals is an immediate generalization of the *potential* with respect to a pair of atoms as it was introduced in Definition 3.9. New is the capability of *adding* argument positions of nonselected atoms to the "partition." A measure function with *external* potential can be refined into a decreasing one, doing just that. At this point, we will not include explicit generalizations of Definitions 3.10 and 3.11. Instead, we immediately define an extended notion of *tight decreasing refinement.*

Definition 4.8. Let P be a program containing an n-ary predicate symbol p. Let O be a cco-k-partition associated to p in P. Let G_1 and G_2 be two goals in the language underlying P whose selected atom contains p. Let $\|.\|_{p,O}$ have (G_1, G_2)-potential. Then $\|.\|_{p,O'}$ is the *tight* (G_1, G_2)-*decreasing refinement* of $\|.\|_{p,O}$ if O' is defined as follows:

- If $\|.\|_{p,O}$ has *internal* (G_1, G_2)-potential and $O[l]$ is O's rightmost (G_1, G_2)-sensitive component, then:
  - $\forall 1 \leq j < l : O'[j] = O[j]$
  - $\forall l < j \leq k : O'[j+1] = O[j]$
  - If $O[l]$ is a p-component, then:
    - $O'[l] = \{i \in O[l] \mid i$ is (G_1, G_2)-decreasing or (G_1, G_2)-stable$\}$
    - $O'[l+1] = \{i \in O[l] \mid i$ is (G_1, G_2)-increasing$\}$
  - Else:
    - $O'[l] = \{i_q \in O[l] \mid i_q$ is (G_1, G_2)-context-decreasing or (G_1, G_2)-context-stable$\} \setminus \{i_q \mid M(G_1, q, i) = \perp\}$
    - $O'[l+1] = \{i_q \in O[l] \mid i_q$ is (G_1, G_2)-context-increasing$\}$ $\cup \{i_q \mid M(G_1, q, i) = \perp\}$.

- *Else* if $\|.\|_{p,O}$ has *external* (G_1, G_2)-potential, then:
  — $O'[1] = \{i \mid i$ is (G_1, G_2)-context-decreasing and context-absent in $O\}$
  — $\forall 1 \leq j \leq k : O'[j+1] = O[j]$.

Again, the first part of the above definition generalizes Definition 3.12. Notice the special treatment for $\perp$-valued arguments. This is necessary since $\perp$ is an absorbing element for $+$ in $\mathbb{N}_b$. The second part is extra. It can be noted that our particular choice for what might be termed a *tight* (G_1, G_2)*-decreasing external refinement* is, in fact, not so tight, in the sense that a reformulation of the second property in Proposition 3.4 does not hold. Several "tighter" variants can be imagined, but we do not believe they would significantly improve the behavior of Algorithm 4.2 on a meaningful class of programs. In consequence, we have preferred intuitive appeal and simplicity of formulation. We do have the following property:

Proposition 4.2. Let P, p, O, G_1, G_2, and O' be as in Definition 4.8. Then $\|.\|_{p,O'}$ is (G_1, G_2)-decreasing.

PROOF.

- The result is immediate when O' is an *external* refinement.
- The proof for the *internal* refinement case is similar to the proof for point (1) in Proposition 3.4. We just need the extra observation that any (G_1, G_2)-context-decreasing argument i_q in $O[l]$ (in case it is a non-p-component) certainly has $M(G_1, q, i) \neq \perp$, thus guaranteeing that $O'[l]$ is nonempty and (G_1, G_2)-decreasing. □

We can now formulate a first algorithm for automatic maximal sensible unfolding under some computation rule preferences. The algorithm assumes that a computation rule R is partially specified in the sense that, for any goal, a (partial) priority order among atoms, candidate for selection, is known (see, e.g., Example 4.6 above).

Algorithm 4.2

Input
a definite program P
a definite goal $\leftarrow A$
(and a (partially specified) computation rule R)
Output
a finite SLD-tree τ for $P \cup \{\leftarrow A\}$
Initialization
$\tau := \{(\leftarrow A, 1)\}$ {* *an SLD-tree with a single derivation* *}
$Pr := \emptyset$ {* *in Pr, the $>_{pr}$-relation will be constructed* *}
$Terminated := \emptyset$
$Failed := \emptyset$
For each recursive predicate p/n in P : $O_p := (\{1, \ldots, n\})$
{* *We set out with cco-1-partitions*
grouping all p's argument positions in a single p-component
and without any non-p-components *}
While *there exists a derivation D in τ such that $D \notin Terminated$* **do**
Let (G, i) name the leaf of D

If $(G, i) = (\square, i)$
Then {* (G, i) *is a success node* *}
add D *to Terminated*
Else
{* *First, we try to determine* $R(G, i)$ *} *Select the leftmost R-preferred atom* $p(t_1, \ldots, t_n)$ *in* G *such that one of the following (mutually exclusive) conditions is satisfied:*
- (G, i) *has no direct covering ancestor*
- (G', j) *is the direct covering ancestor of* (G, i) *and* $\|.\|_{p,O_p}$ *is* (G', G)*-decreasing*
- (G', j) *is the direct covering ancestor of* (G, i) *and* $\|.\|_{p,O_p}$ *has* (G', G)*-potential and* $\|.\|_{p,O'_p}$ *is its tight* (G', G)*-decreasing refinement*

If *such an atom* $p(t_1, \ldots, t_n)$ *cannot be found*
Then {* (G, i) *becomes a dangling leaf* *}
Add D *to Terminated*
Else
$R(G, i) := p(t_1, \ldots, t_n)$
If $R(G, i)$ *was selected on the basis of the third condition above*
Then {* *Register the new cco-partition* *}
$O_p := O'_p$
Let $Resolvents(G, i)$ *name the set of all derivation steps that can be performed*
If $Resolvents(G, i) = \emptyset$
Then {* (G, i) *is a failure node* *}
Add D *to Terminated and Failed*
Else
{* *Extend the derivation* *} *Expand* D *in* τ *with the elements of* $Resolvents(G, i)$ *Let* $Descend(R(G, i), i)$ *name the set of all pairs* $((R(G, i), i), (B\theta, j))$*, where*
— B *is an atom in the body of a clause applied in an element of* $Resolvents(G, i)$
— θ *is the corresponding mgu*
— j *is the number of the corresponding descendant of* (G, i)
Apply θ *to the affected elements of* Pr *Add the elements of* $Descend(R(G, i), i)$ *to* Pr
Endwhile

Theorem 4.2. Algorithm 4.2 terminates. The resulting SLD-tree τ *is finite.*

PROOF. The only substantially new element is the facility to externally refine a cco-partition. It can be seen that this does not jeopardize termination since:

- A program P contains only a finite amount of recursive predicates, all of finite arity.
- Any argument position of any recursive predicate can occur in at most one nonselected atom component of a cco-partition, and once included in the partition, is never removed. □

Example 4.7. Applying Algorithm 4.2 to the programs, algorithms, and computation rules discussed in Examples 4.1, 4.5, and 4.6 produces the results presented there.

4.3. Backpropagation of Instantiations

At this point, it is interesting to look at the behavior of Algorithm 4.2 when *no* computation rule preferences are, in fact, specified. In that case, the search for an unfoldable atom reduces to the basic "take the leftmost whose (possibly refined) measure function allows it" technique, used for automation in [8] and Section 3 above. It turns out that, thanks to its context-considering capabilities, Algorithm 4.2 can often improve upon an annoying deficiency of algorithms solely focusing on selected atoms. Consider the following typical example.

Example 4.8.

$bp(X,Y) \leftarrow a(X,Z), b(Z,Y)$
$a([],Y) \leftarrow$
$a([X \mid Xs],Y) \leftarrow do\_a(X,Y), a(Xs,Y)$
$b([],[]) \leftarrow$
$b([X \mid Xs],[X \mid Ys]) \leftarrow b(Xs,Ys)$

(The definition for *do_a* does not matter here. Just suppose it can be fully resolved at unfolding time.)

We are interested in unfolding the following query:

$\leftarrow bp(X,[1 \mid Ys]).$

Part of the SLD-tree generated by Algorithm 4.2 is depicted in Figure 5.

The unfolding carried out in node (**) is particularly interesting. At that point, $\|\cdot\|_{a,(\{1,2\})}$ is increasing with respect to the pair formed by (**) and its direct covering ancestor (*). Moreover, it has no internal potential. But $\|\cdot\|_{a,(\{2_b\},\{1,2\})}$ is its *external* (*, **)-decreasing refinement. It maps (*) into $(1,0)$ and (**) into $(0,1)$, and therefore allows unfolding the a-goal. No measure function solely based on a-arguments can do likewise.

The above example illustrates a general phenomenon: in unfolding, back- (i.e., in the reverse direction as the "scan" for unfoldable atoms) propagation of information can be considered similar to coroutining. (Observe that a further instantiated second argument in the *bp* starting goal would lead to a further little by little passing of information chunks from the b to the a goal.) Therefore, the work in this section also improves on earlier algorithms with respect to this issue. We include a final example, illustrating that the mere capacity to register the "disappearance" of atoms from the context already has beneficial effects.

Example 4.9. Consider the well-known "naive reverse" program:

$rev([],[]) \leftarrow$
$rev([X \mid Xs],Y) \leftarrow rev(Xs,Z), app(Z,[X],Y)$
$app([],X,X) \leftarrow$
$app([X \mid Xs],Y,[X \mid Zs]) \leftarrow app(Xs,Y,Zs)$

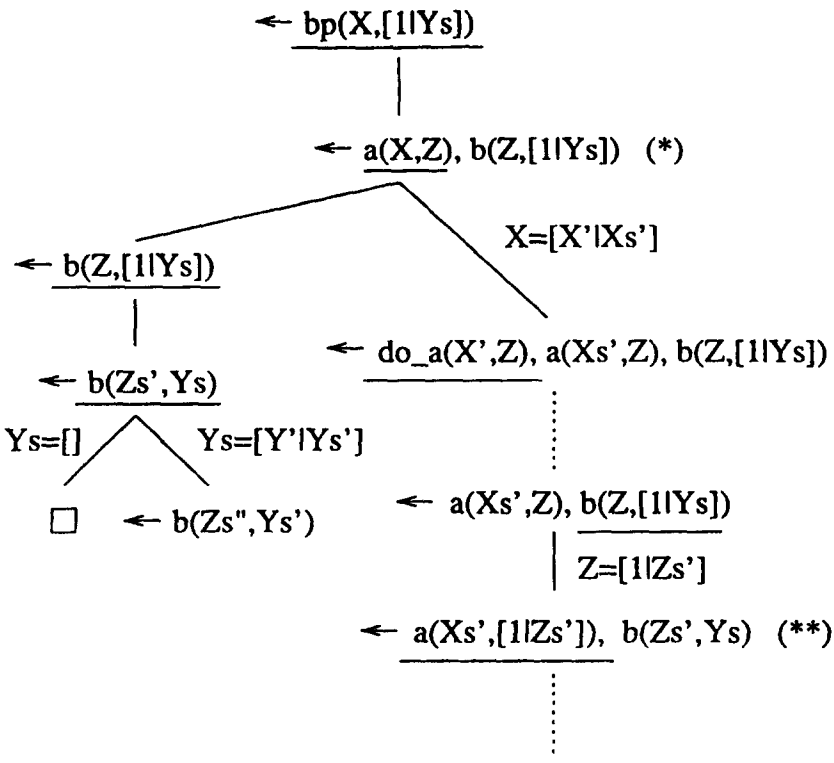

FIGURE 5. Handling backpropagation.

together with the following query:

$$\leftarrow rev([1 \mid Xs], Y).$$

In the context of partial deduction (see Section 2.1), from an SLD-tree produced by (basically) the unfolding method in [8], the following clauses can be synthesized:

$$rev([1],[1]) \leftarrow$$
$$rev([1, X \mid Xs],[X,1]) \leftarrow rev(Xs,[])$$
$$rev([1, X \mid Xs],[Y,1]) \leftarrow rev(Xs,[Y \mid Ys]), app(Ys,[X],[])$$
$$rev([1, X \mid Xs],[Y,Z \mid Zs]) \leftarrow rev(Xs,[Y \mid Ys]),$$
$$app(Ys,[X],[Z \mid Zs']), app(Zs',[1],Zs)$$

Using Algorithm 4.2 for unfolding, applying a simple "select the leftmost suitable atom" computation rule, we obtain the more sensible

$$rev([1],[1]) \leftarrow$$
$$rev([1,X],[X,1]) \leftarrow$$
$$rev([1, X \mid Xs],[Y,Z \mid Zs]) \leftarrow rev(Xs,[Y \mid Ys]),$$
$$app(Ys,[X],[Z \mid Zs']), app(Zs',[1],Zs)$$

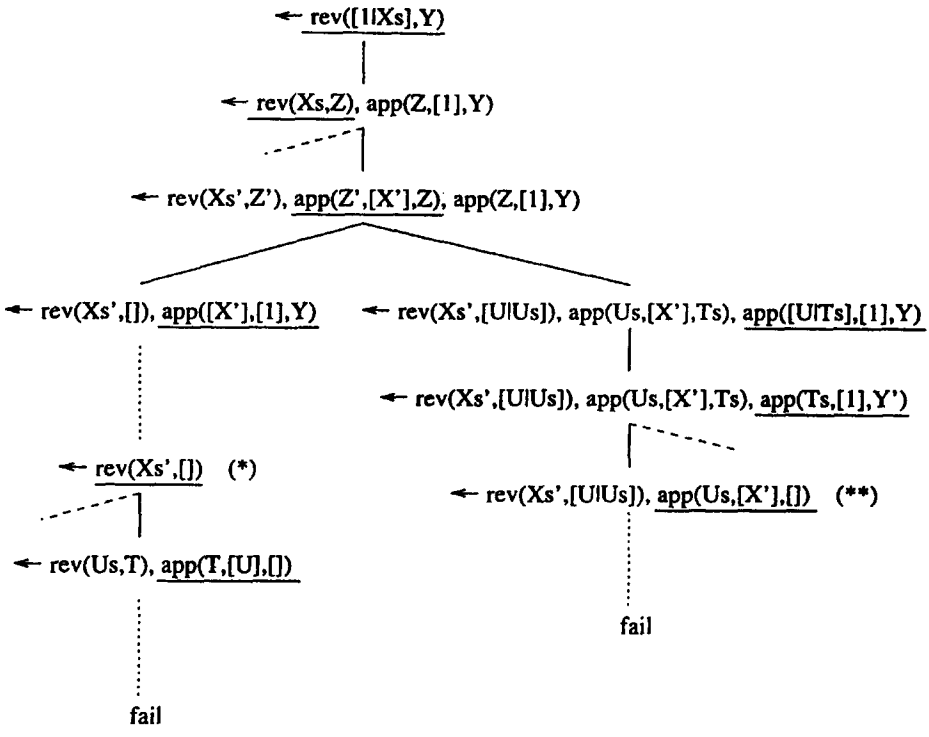

FIGURE 6. Part of the SLD-tree generated for Example 4.9.

The resulting cco-partitions are

O_{rev}=$(\{1_{app}, 2_{app}, 3_{app}\}, \{1, 2\})$

O_{app}=$(\{1_{app}, 2_{app}, 3_{app}\}, \{1, 2, 3\})$

We do not include the complete generated SLD-tree, but an interesting portion can be found in Figure 6. Nondepicted branches are indicated by a dashed link at their origin. Particularly noteworthy are the unfoldings at node (*) and node (**): they cause the differences between the first and second sets of resulting clauses above.

Summarizing, we can state that we generalized the methods for weight-based finite unfolding by allowing (also) the consideration of nonselected atoms in a goal. We have presented two algorithms incorporating this feature, the second one automatically focusing on sensible measure functions. We have shown how they handle coroutining and the related issue of instantiation backpropagation.

Numerous variants of and/or enhancements to Algorithm 4.2 can be considered.

- One can include arguments of nonrecursive context atoms in cco-partitions.
- Priority can be given to unfolding atoms which cause little branching in the tree.
- Using separate cco-partitions for different chains of covering nodes seems reasonable, and will probably have a more profound influence than is the case in approaches solely based on measuring selected atoms.

- It seems possible that, in larger applications, some extra limitation should be imposed, restricting unfolding atoms on account of a contextual weight decrease to those that have actually been "influenced" by those context unfoldings. It is, however, not immediately clear how to pin down this notion of "influenced." Demanding a higher weight, or simply not being a variant of the call selected in the direct covering ancestor, are too restrictive conditions, as an inspection of the coroutining Examples 4.1 and 4.6 reveals. A comparison with the state of the corresponding atom in some intermediate goal node seems more appropriate.
- A further step towards more sophistication can involve a static, *off-line analysis* phase, scrutinizing the program code.[2] In this way, one can probably derive useful supporting information about (mutual) influences among argument positions. This might help to resolve the problem indicated above. It might also lead to more "sensible" tight decreasing refinements. Reconsider Example 4.9 above. Rather than the resulting O_{rev}, we might consider the partition $(\{1\}, \{1_{app}\}, \{2\})$ (or $(\{1\}, \{1_{app}, 2_{app}, 3_{app}\}, \{2\})$) as the natural one to be used for the *rev* predicate. In fact, with these partitions, the whole tree is subsetwise founded, while this is not the case with O_{rev}. The development of relevant analysis techniques and an assessment of their value are challenging topics for future research.

Rather than delving into the above-sketched issues, in the next subsection, we will address another topic glossed over by the development so far. Indeed, both Algorithms 4.1 and 4.2, relying on Definition 4.5, are incapable of distinguishing between several nonselected atoms with the same predicate symbol in one goal. This is fine for most producer–consumer coroutining applications, where the basic structure is typically of the simple linearly recursive type as exhibited by the programs in Examples 4.1 and 4.6. But, in general, they might fall short of dealing properly with instantiation backpropagation in logic programs exhibiting a more complex structure.

4.4. Focusing on Ancestor Atoms

We restrict ourselves to a somewhat informal discussion of the issue and its most striking aspects, omitting a complete technical development.

Example 4.10. Let us unfold the naive reverse program with respect to the following query:

$\leftarrow rev([1, 2 \mid Xs], Y)$

using the cco-partitions produced in Example 4.9. Some fragments from an interesting SLD-derivation are shown in Figure 7.

In (*), no atom can be selected for unfolding. Our measure functions are too coarse; they do not register the "disappearance" of the last *app* atom.

A solution for the problem illustrated in Example 4.10 is obvious: introduce more fine-grained comparisons among context atoms. The second part of Definition 4.5 does not distinguish between atoms in different ancestor–descendant chains.

[2] In fact, the algorithms included in this paper presuppose a very simple analysis of this kind: determining which predicates in the given program are recursive.

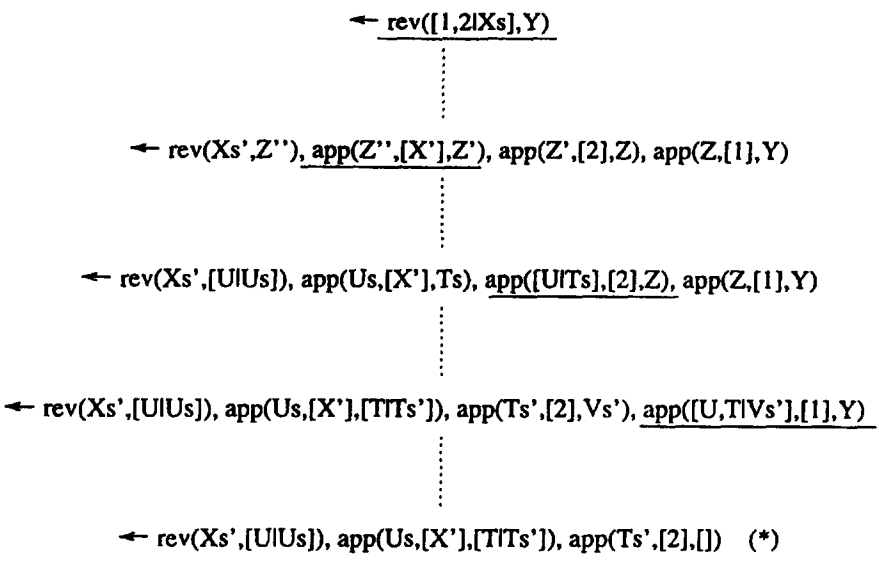

FIGURE 7. A case for yet more powerful measures.

Redefining cco-partitions and their associated measure functions in this sense is possible: compare (maximum) argument weights of context atoms in the descendant goal with the argument weights of the corresponding atom in the ancestor goal, situated on the same branch of the associated proof tree. (Of course, often the comparison will simply be between an atom and a possibly less instantiated version of the same atom. This is the case when the considered atom itself has not been the subject of unfoldings carried out between the two inspected goals.) It can be noted that such an approach would indeed allow further unfoldings in Example 4.10.

Rather than going through the complete technical development, rephrasing definitions and results in Section 4.2, we conclude this section by pointing out an intriguing additional difficulty emerging in this context. Consider the following schematic example.

Example 4.11. Suppose we unfold the program:

$a(X) \leftarrow a(Y), b(X, Y)$

$b(X, X) \leftarrow$

some other (recursive) clauses for b

and the query:

$\leftarrow a([1 \mid X])$

using more refined measure functions of the kind sketched above. Part of a possible SLD-derivation is depicted in Figure 8. The unfolding in $(**)$ is allowed because of b's disappearance compared with $(*)$. In this way, continuously appearing and disappearing "fresh" b atoms create an infinite series of tight decreasing refinements. As a result, *unfolding does not terminate.*

Example 4.11 shows that an unrestricted application of fine-grained context considering unfolding techniques may lead to nontermination. Imposing a *bound on*

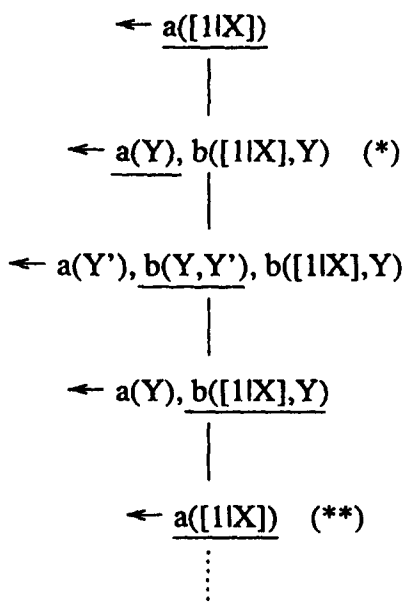

FIGURE 8. An infinite SLD-derivation.

the number of components allowed in a cco-partition is an obvious remedy. More refined variants of this basic idea can, of course, be imagined, e.g., predicatewise bounds. Moreover, it seems likely that *off-line analysis* can be helpful, perhaps even to the extent of reducing the choice of measure functions to a restricted range, again safely guaranteeing termination.

However, we will leave these and related considerations as a subject for future research. Indeed, Example 4.11 also sheds further light on the *principles underlying automatic finite unfolding.* It is this latter issue that, in the next section, we wish to reconsider in proper detail. Meanwhile, we believe that Sections 3 and 4, as they stand, give a good impression of the power and generality of partition-based unfolding, using lexicographical priorities among arguments occurring in goals.

5. REFINING MEASURE FUNCTIONS: A GENERIC TREATMENT

5.1. Introduction

Now that we have worked our way through quite a few algorithms, a clarified picture of common underlying principles emerges. This section is devoted to a formalization of that understanding.

First, we notice that the framework laid out in Section 2.2 has proved to be quite general. We were able to deal both with Example 3.2 and Examples 4.5 and 4.6 using suitably specialized versions of Algorithm 2.1. In the next section, we *will* find occasion to relax some of its inherent limitations. For the moment, however, we concentrate on our increased insight in the structure of automation.

Compared with Algorithm 2.1, fully automatic algorithms presented or mentioned in [8, 37] and Sections 3.3, 3.4, 4.2.2, and 4.4:

- Dynamically fix the computation rule, possibly keeping count of statically fixed preferences among selectable atoms.

- All use the same particular recipe for choosing $R_0, R_1, \ldots, R_N$, assigning one R_i per recursive predicate symbol in the program, to remain unchanged throughout the execution of the algorithm.
- Each incorporate one particular basic strategy to assign to a goal a weight, i.e., an element in some well-founded (until now always totally ordered) set.
- Use measure functions based on this strategy, one function per R_i, to associate concrete weights to goals. Initial choices for these measure functions are built into the algorithms. Measure functions are refined dynamically, if desirable and possible, to enable further unfolding.

In this section, we will concentrate on a generic formalization of the latter point, treating the other issues at about the same level of generality as was adhered to in Section 2.2.

5.2. A Generic Algorithm

We start with some definitions.

Definition 5.1. Let P be a definite program; then we denote by $Atom_P$ the set of atoms that can be formed in the language underlying P.

Definition 5.2. Let P be a definite program, $\mathcal{L}_P$ its underlying language, and suppose that $R_0, R_1, \ldots, R_N$ is a partition of $Atom_P$. Let $W, >_W$ be some well-founded set. Then we call a function

$$F : \{G \mid G \text{ is a definite goal in } \mathcal{L}_P \text{ with selected atom } \in R_k\} \to W, >_W$$

an (R_k, P)-*applicable measure function with target set* W.

Definition 5.3. A set $\{F \mid F \text{ is an } (R_k, P)\text{-applicable measure function}\}$ is called an (R_k, P)-*applicable measure space.*

In the sequel, when P (or $\mathcal{L}_P$) is clear from the context, we will often denote an (R_k, P)-applicable measure space by $\mathcal{F}_k$. Note that different elements of a measure space may have different target sets. Finally, we demand that some *(partial) order relation* be defined on measure spaces. For a space $\mathcal{F}_k$, we will denote it by $\gg_k$.

We can now formulate a generic algorithm for automatic, weight-based, finite unfolding. In order to produce an executable instance of it, the following are necessary:

- Computation rule preferences can be stipulated, resulting in a concrete meaning of the term "R-preferred." If none is given, all atoms are equally R-preferred.
- $R_0, R_1, \ldots, R_N$ should be chosen such that they form a partition of $Atom_P$ and guarantee the satisfaction of the third condition in Definition 2.9.
- For every k, $1 \leq k \leq N$, we must specify $\mathcal{F}_k, \gg_k$ and choose an initial F_k in $\mathcal{F}_k$.

We will call any executable instance of Algorithm 5.1, thus determined, *proper.*

Algorithm 5.1

Input
a definite program P
a definite goal $\leftarrow A$
Output
an SLD-tree τ *for* $P \cup \{\leftarrow A\}$
Initialization
τ *is initialized as the SLD-tree that contains a single derivation, consisting of the goal* $\leftarrow A$, *without selected atom.*
Initial choices are made for the measure functions $F_1, \ldots, F_N$.
While *there exists a nonterminated derivation* D *in* τ **do**
If D *is successful,* **Then** *terminate* D
Else If D*'s leaf-node contains no selectable atom,* **Then** *terminate* D
Else
select an R-preferred selectable atom
If *no derivation steps are possible,* **Then** *terminate and fail* D
Else *extend* D
Where *an atom* $p(t_1, \ldots, t_n) \in R_k$ *in a goal* G *is selectable if one of the following (mutually exclusive) conditions holds, in case it is actually selected:*
- G *has no direct covering ancestor*
- G'*is the direct covering ancestor of* G *and* $F_k(G') >_k F_k(G)$
- G' *is the direct covering ancestor of* G *and not* $(F_k(G') >_k F_k(G))$ *and* $\exists F \in \mathcal{F}_k$: (1) $F_k \gg_k F$ (2) $F(G') >_k F(G)$

If *an atom* $p(t_1, \ldots, t_n)$ *has been selected on the basis of the third condition,* **Then** *replace* F_k, *in the set of measure functions in use, by some* F'_k *satisfying the conditions (1) and (2) above.*
Endwhile

We have the following theorem:

Theorem 5.1. A proper instance of Algorithm 5.1 terminates for a definite program P *and goal* $\leftarrow A$, *producing a finite SLD-tree* τ *for* $P \cup \{\leftarrow A\}$, *if* $\forall k, 1 \leq k \leq N : \mathcal{F}_k, \gg_k$ *is a well-founded set. The resulting* τ *is subsetwise nearly founded with respect to the final* $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$.

PROOF. The well-foundedness condition on the (finitely many) measure spaces ensures that a change in the set of measure functions used can occur only finitely many times. After the last such change, the measure functions corresponding to the final $(F_1, \ldots, F_N)$

- might be nonmonotonic on the finite subtree constructed thus far
- but any proper instance of Algorithm 5.1 will from then on function as an instance of Algorithm 2.1.

The result follows. □

It can be verified that [37, Algorithm 3.6] and Algorithms 3.2 and 4.2 in the present paper are proper instances of Algorithm 5.1, and therefore terminate. Indeed:

- Only Algorithm 4.2 considers user-specifiable computation rule preferences; the others do not. All eliminate remaining nondeterminism in atom selection through a "choose the leftmost selectable" strategy.
- The choice for $R_0, R_1, \ldots, R_N$ is always along the same basic line: one R_i per recursive predicate symbol in P, R_0 for atoms featuring a nonrecursive predicate symbol. Obviously, this satisfies the above-specified conditions on $R_0, R_1, \ldots, R_N$.
- The measure spaces used are of increasing complexity:
  - [37, Algorithm 3.6] uses set-based measures, universally mapping to $\mathbb{N}, >$.
  - The measure functions in Algorithms 3.2 and 4.2 above are, respectively, based on partitions and cco-partitions of argument positions. They map to some $\mathbb{N}^i, \succ_i$ or $\mathbb{N}^i_b, \succ_i$, respectively, i varying within measure spaces.
- Definition 3.11 can serve as a basis for the required order relation on partition-based measure spaces. Indeed, transitively closing the inverse of the "is a refinement of" relation results in a strict partial order on such spaces. Introducing a similar notion for set- and cco-partition-based measures is straightforward and implicit in the work presented.
- *All measure spaces involved are finite, and therefore (trivially) well-founded.*
- Remaining nondeterminism in the choice of a refined, properly decreasing measure function (i.e., satisfying conditions (1) and (2) in Algorithm 5.1) has always been removed by imposing extra conditions (see, e.g., Definition 3.12 and Proposition 3.4).
- Finally, some bookkeeping instructions related to the maintenance of the covering relationship have not been included in the generic Algorithm 5.1.

Algorithm 3.1 is identical to Algorithm 3.2, except for the presence of condition (*) in the former. Obviously, this extra condition does not damage termination. It just ensures subsetwise foundedness of the resulting SLD-tree, a special case of subsetwise nearly foundedness, where the final measure functions are subsetwise monotonic. Next, the automatic version of [8, Algorithm 3.2] also imposes the subsetwise foundedness check. In spite of some further technical differences with the current development, its termination behavior is amenable to basically the analysis presented here.

Finally, it is interesting to reconsider Section 4.4. The underlying reasons for nontermination in Example 4.11 now become clear:

1. Measure spaces are no longer guaranteed to be finite.
2. The adapted notion of refinement no longer guarantees that the resulting ordering is well-founded.

Imposing a bound on the number of components reestablishes finiteness, while more sophisticated order relations, inspired by an off-line program analysis, could perhaps guarantee the well-foundedness of measure spaces, left infinite.

6. DEALING WITH DATALOG, INCORPORATING VARIANT CHECKING

6.1. Introduction

Structure-based weights, as presented above, are obviously not a good basis to control unfolding of datalog (i.e., functor-free) programs. Consider the following example, borrowed from [1].

Example 6.1.

$reach(X, X) \leftarrow$

$reach(X, Z) \leftarrow reach(X, Y), edge(Y, Z)$

$edge(a, b) \leftarrow$

$edge(b, d) \leftarrow$

$edge(f, g) \leftarrow .$

For the query

$\leftarrow reach(a, X)$

partial deduction produces the following specialized clauses from the SLD-tree built according to any of the methods discussed above:

$reach(a, a) \leftarrow$

$reach(a, b) \leftarrow reach(a, a)$

$reach(a, d) \leftarrow reach(a, b)$

$reach(a, g) \leftarrow reach(a, f).$

If, instead, we *unfold the leftmost atom without an unfolded variant* on the same derivation, we are able to derive

$reach(a, a) \leftarrow$

$reach(a, b) \leftarrow$

$reach(a, d) \leftarrow.$

However, for logic programs with functors, this nonvariant-based unfolding, in general, does not guarantee termination. We include the following example:

Example 6.2. Consider the well-known list reverse program with accumulating parameter:

$rev([], X, X) \leftarrow$

$rev([X \mid Xs], Y, Z) \leftarrow rev(Xs, [X \mid Y], Z)$

and the query:

$\leftarrow rev([1, 2 \mid Xs], [], Z).$

The infinite SLD-tree generated by nonvariant-based unfolding for this example is depicted in Figure 9. Along the rightmost branch of the tree, *rev*'s second argument

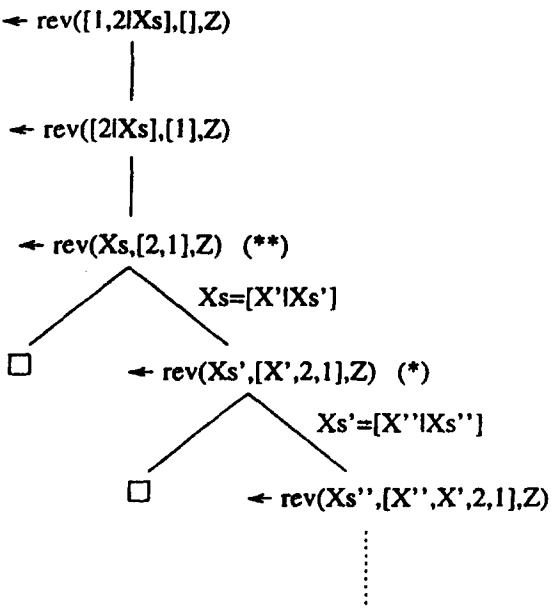

FIGURE 9. Infinite unfolding using the nonvariant rule.

grows with each unfolding. In this way, an infinite series of atoms is produced, none of which is a variant of any other.

One possible solution can be identifying datalog predicates in a program. Abstract interpretation (see, e.g., [9]) for example, should be able to uncover predicates whose arguments will certainly be free from functors. These can then be treated in a special way during unfolding, ignoring their (0) weight, but applying the above-mentioned nonvariant check. However, we have opted for a more global approach, unifying the two criteria in one generally applicable unfolding methodology. The basic idea can be described as follows: selecting an atom A in a goal G is permitted, even when this results in *equal weights* for G and its direct covering ancestor, if no goal covering G has a selected atom which is a variant of A. However, it is clear that a straightforward application of this coarse rule does not always guarantee finiteness. Let us reconsider the above example.

Example 6.3. Under the set-based measure function $|.|_{rev,\{1,3\}}$, all nonempty goals from the third downwards in Figure 9 have weight 0, and yet selected atoms are not variants. However, restricting the attention to the first and third argument, i.e., those "measured" by $|.|_{rev,\{1,3\}}$, we find that they *are* variants.

Example 6.3 shows that welding together weight- and nonvariant-based unfolding requires some care, but it also suggests that the enterprise is not hopeless. The rest of this section formally develops this issue. Its layout follows the meanwhile familiar pattern: we first adapt/specialize the basic conceptual framework underlying Algorithm 2.1. In a second step, we address full automation.

6.2. Reconsidering the Framework

Throughout this section, we will be dealing with definite goals containing a finite number of atoms. This allows us to suppose a numbering on the atoms

in a goal, e.g., from left to right, starting with 1, thus determining an argument position in a goal by a pair of natural numbers: (*atom_number_within_goal*, *argument_number_within_atom*).

In order to simplify the ensuing presentation, we agree on the following for the rest of Section 6.2:

- Given a definite program P, $\mathcal{L}_P$ will denote its underlying language. *We assume that $\mathcal{L}_P$ contains at least one function symbol*, even if none occurs in P itself. The reason for this slight deviation from convention will become clear below.
- A goal to which some measure function F can be applied will be called *F-suitable*. For instance, a goal in some language $\mathcal{L}_P$, such that $R_0, R_1, \ldots, R_N$ is a partition of $Atom_P$, is suitable for some (R_k, P)-applicable measure function if its selected atom is in R_k.

We can now introduce the following:

Definition 6.1. Let F be a measure function and G an F-suitable goal. Then we call an argument position (i, j) of G *F-measured* if there exists an F-suitable goal G' such that:

1. $F(G') \not\equiv F(G)$
2. G' and G are identical, except at argument position (i, j).

We call the set of all F-measured argument positions of G its *F-measured argument set*, denoted $MS_F(G)$, and the multiset of terms occurring on these argument positions in G its *F-measured part*, denoted $MP_F(G)$. Finally, we will use the notation $t_G(i, j)$ to denote the term corresponding to an argument position (i, j) in a goal G.

The intuition behind this definition is simple: measured argument positions indicate terms whose structure influences the weight associated to the goal. The reason for demanding the presence of at least one functor in the language now becomes clear: in a language without any function symbols, the first condition above would be unsatisfiable for the structure-based measure functions in the focus of our interest. Definition 6.1 in that case no longer correctly formalizes the intuitive notion just described.

Example 6.4.

- With a constant measure function, mapping any suitable goal to the same element in its target set, goals obviously have empty measured argument sets.
- A measure function, merely counting the number of atoms (possibly containing a given predicate symbol) in a goal, likewise results in empty measured argument sets.
- A suitable goal's measured argument set under a *set-based* measure $|.|_{p,S}$ contains exactly the argument positions corresponding to the selected atom's argument positions in S.
- All argument positions of the selected atom, and none other, are in the measured argument set under a *partition-based* measure.

- Finally, all argument positions corresponding to elements occurring in the underlying cco-partition are in a goal's measured argument set under a *cco-partition-based* measure function. Consider, e.g., the second goal from the top, included in Figure 7. Let O_{app} be $(\{1_{app}, 2_{app}, 3_{app}\}, \{1,2,3\})$. Then that goal has the following $\|.\|_{app,O_{app}}$-measured argument set: $\{(i,j) \mid 2 \leq i \leq 4, 1 \leq j \leq 3\}$.

We will be interested in verifying whether the measured parts of two goals are "variants" of each other. The following definition lays down an exact content of the variant notion in this specialized context.

Definition 6.2. Suppose F is a measure function, and let G and G' be two F-suitable goals. Then we say that *the F-measured parts of G and G' are variants*, denoted $MP_F(G) \sim MP_F(G')$, iff there is a one-to-one correspondence C between $MS_F(G)$ and $MS_F(G')$ such that $((i,j),(i',j')) \in C$ implies:

- $j = j'$.
- The ith atom in G and the i'th atom in G' have the same predicate symbol.
- The ith atom in G is selected iff the i'th atom in G' is.
- There exist (renaming) substitutions θ and θ' such that

$$\forall((i,j),(i',j')) \in C : t_G(i,j)\theta = t_{G'}(i',j') \wedge t_{G'}(i',j')\theta' = t_G(i,j).$$

We will also use the following notation: $G \sim_F G'$. Obviously, $\sim_F$ is an equivalence relation on the set of F-suitable goals.

The first three items above demand "equality" of the two measured argument sets. In that case, we can properly compare the terms in the corresponding measured parts, and require that they indeed be variants, as expressed in the fourth item.

Example 6.5. Consider the following goals (selected atoms are underlined):

- $G = \leftarrow \underline{p(f(X),X)}, q(g(Y))$
- $G' = \leftarrow q(g(X')), q(g(Y')), \underline{p(f(Z'),Z'')}$

and the following measure functions:

- $F_1 = |.|_{p,\{1\}}$
- $F_2 = |.|_{p,(\{1\},\{2\})}$
- $F_3 = \|.\|_{p,(\{1_q\},\{1,2\})}$.

Then we have:

- $MS_{F_1}(G) = \{(1,1)\}$
- $MS_{F_1}(G') = \{(3,1)\}$
- We can take: $C = \{((1,1),(3,1))\}$, $\theta = \{X/Z'\}$, $\theta' = \{Z'/X\}$. And therefore: $G \sim_{F_1} G'$.

And:

- $MS_{F_2}(G) = \{(1,1),(1,2)\}$
- $MS_{F_2}(G') = \{(3,1),(3,2)\}$
- $C = \{((1,1),(3,1)),((1,2),(3,2))\}$ (uniquely) satisfies the first three conditions in Definition 6.2. But no θ and θ', as required in the last part of the above definition, can be constructed.
  So: $G \not\sim_{F_2} G'$

Finally:

- $MS_{F_3}(G) = \{(1,1),(1,2),(2,1)\}$
- $MS_{F_3}(G') = \{(1,1),(2,1),(3,1),(3,2)\}$
- There is no one-to-one correspondence between $MS_{F_3}(G)$ and $MS_{F_3}(G')$. This means: $G \not\sim_{F_3} G'$.

Notice that all three measure functions assign equal weights to G and G': 1, $(1,0)$ and $(1,1)$, respectively.

We can now adapt Algorithm 2.1 (we only include an updated version of its main loop):

Algorithm 6.1

While *there exists a nonterminated derivation* $D \in \tau$ **do**
Let (G,i) *be the leaf of* D
Let Resolvents(G,i) *be the set of all its direct* $>_{\tau_0}$*-descendants*
If *Resolvents*$(G,i) = \emptyset$
Then *terminate* D
Else if *there is a direct covering ancestor* (G',j) *of* (G,i) *with* $R(G',j)$, $R(G,i) \in R_n$ *such that none of the following is satisfied:*
1) $F_n(G',j) >_n F_n(G,i)$
2) $F_n(G',j) \equiv F_n(G,i) \wedge$
$\neg\exists(G'',k) \in D : (G'',k)$ *covers* $(G,i) \wedge F_n(G'',k) \equiv F_n(G,i) \wedge$
$(G'',k) \sim_{F_n} (G,i)$
Then *add* D *to Terminated*
Else
Replace τ *by* $\tau \setminus D \cup \{D \cup \{(G^*,l)\} \mid (G^*,l) \in Resolvents(G,i)\}$
Endwhile

Algorithm 6.1 differs from Algorithm 2.1 by the presence of condition 2). It allows extending a derivation whose leaf has a weight identical to the weight of its direct covering ancestor, but contains a measured part which is not a variant of the measured part found in any covering ancestor of equal weight.

Example 6.6. Consider the program and query in Example 6.1. Take:

- $R_0 = \{$atoms containing *edge*$\}$
- $R_1 = \{$atoms containing *reach*$\}$
- $F_1 = |\cdot|_{reach,\{1,2\}}$.

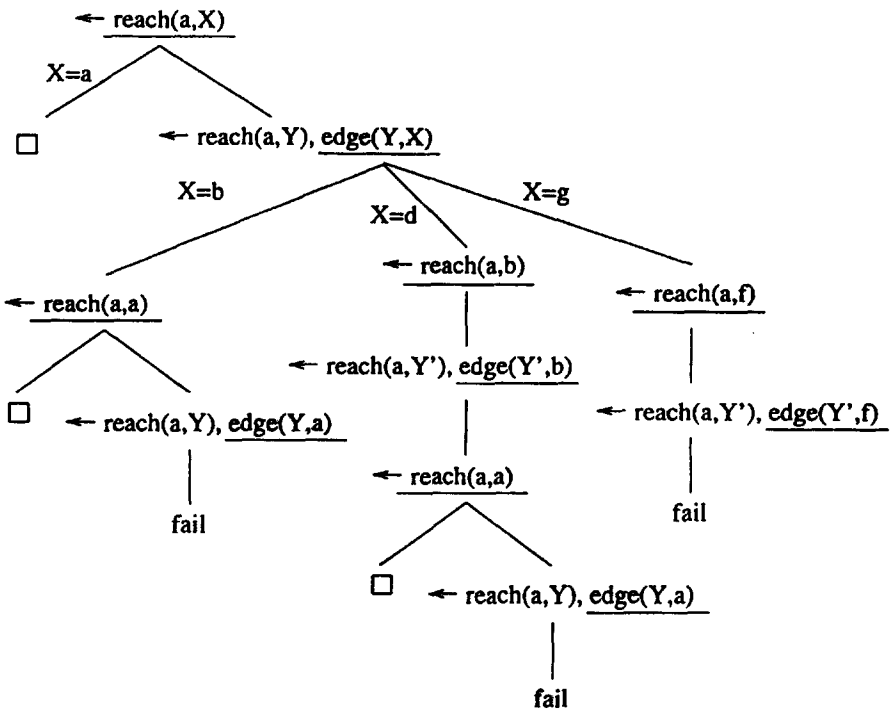

FIGURE 10. A properly unfolded datalog tree.

The SLD-tree produced by Algorithm 6.1 can be found in Figure 10. Notice that all measured goals have the same weight: 0. However, condition 2) in Algorithm 6.1 enables proper unfolding.

Example 6.7. For the program and query treated in Example 6.2, Algorithm 6.1 stops unfolding after the first branching both when applied with $|.|_{rev,\{1,3\}}$ and $|.|_{rev,(\{1,3\},\{2\})}$. In the former case, goal $(*)$ and its direct covering ancestor, $(**)$, have equal weights (0), but also variant measured parts, $\{Xs', Z\}$, and $\{Xs, Z\}$, respectively. $|.|_{rev,(\{1,3\},\{2\})}$, on the other hand, is simply $(**, *)$-increasing.

Finally, we address the question of whether Algorithm 6.1 always terminates for any choice of measure functions. The following example shows that this is not the case.

Example 6.8. Consider the following program fragment:

$$p(X) \leftarrow q(f(X)), r(g(X))$$
$$q(X) \leftarrow q(X), r(X)$$

and unfold the goal

$$\leftarrow p(a)$$

using the cco-partition-based measure function $\|.\|_{q,(\{1_r\},\{1\})}$. The first few nodes of the resulting infinite SLD-derivation are depicted in Figure 11. Measured nodes are annotated with their weight. Note that the measure function is stable, but the

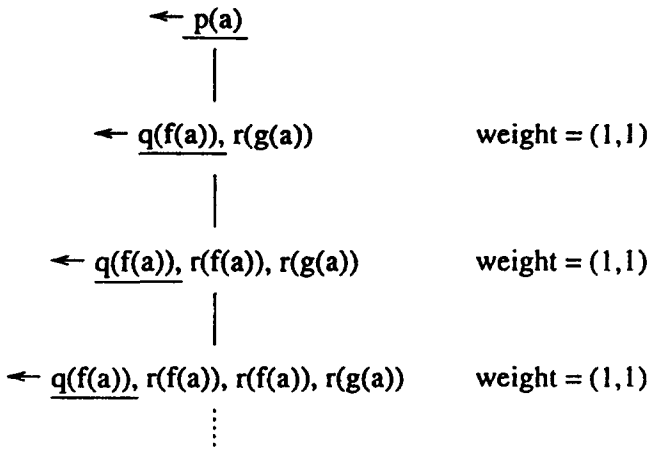

FIGURE 11. Infinitely many goals with equal weight but nonvariant measured part.

measured argument set changes with each unfolding. Condition 2) in Algorithm 6.1 is therefore always satisfied and the derivation never terminated.

So, Algorithm 6.1 may fail to terminate. We need a notion to characterize "safe" measure functions.

Definition 6.3. A measure function F with target set W is called *finitely measuring* for a language $\mathcal{L}$ if, for any weight $w \in W$, the quotient set $\{G \text{ in } \mathcal{L} \mid F(G) \equiv w\}/\sim_F$ is finite.

In other words, only a finite number of goals, expressed in the given language, with nonvariant measured part have the same weight under F.

Definition 6.4. Let P and $\leftarrow A$ be a definite program and goal, respectively. Then $\mathcal{L}_{PA}$ will denote the language determined by the constants, function, and predicate symbols actually occurring in P and A.

Obviously, $\mathcal{L}_{PA}$ is contained in the language underlying P and A, and any goal in an SLD-tree for $P \cup \{\leftarrow A\}$ is in $\mathcal{L}_{PA}$. Observe that (for finite P) $\mathcal{L}_{PA}$ contains only finitely many constant and function symbols. This allows the following theorem:

Theorem 6.1. Algorithm 6.1 terminates for a definite program P and goal $\leftarrow A$, using a given computation rule R and a pair $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$, if $F_1, \ldots, F_N$ are finitely measuring for $\mathcal{L}_{PA}$. The resulting finite SLD-tree τ is subsetwise nearly founded with respect to $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$.

PROOF (Sketch). The proof is similar to the proofs for analogous theorems above, now relying on Definitions 6.3 and 6.4 and condition 2) in Algorithm 6.1, to argue that the measure functions are *nearly* founded on chains of covering nodes. □

Theorem 6.1 shows that Algorithm 6.1 can safely be used with finitely measuring measure functions. The remaining question is whether we can characterize classes of measure functions as finitely measuring. In particular, are measure functions

of the kinds introduced in this paper finitely measuring? We have the following results (overloading function symbols as was implicitly done before):

Proposition 6.1. Let P and $\leftarrow A$ be a definite program and goal, respectively. Let p be a predicate symbol of arity n in P. Let S be a set of argument positions of p, and O an ordered k-partition associated to p. Then:

1. *The measure function*

$$|.|_{p,S} : \{G \mid G \text{ is a definite goal in } \mathcal{L}_P \text{ with selected atom } p(t_1, \ldots, t_n)\} \rightarrow \mathbb{N},$$
$$G \mapsto |p(t_1, \ldots, t_n)|_{p,S}$$

is finitely measuring for $\mathcal{L}_{PA}$.

2. *The measure function*

$$|.|_{p,O} : \{G \mid G \text{ is a definite goal in } \mathcal{L}_P \text{ with selected atom } p(t_1, \ldots, t_n)\} \rightarrow \mathbb{N}^k,$$
$$G \mapsto |p(t_1, \ldots, t_n)|_{p,O}$$

is finitely measuring for $\mathcal{L}_{PA}$.

PROOF. Since $\mathcal{L}_{PA}$ contains only finitely many constant and function symbols, there is only a finite number of nonvariant terms in $\mathcal{L}_{PA}$ to which the term norm $|.|$, defined in Definition 2.13, assigns the same natural number. The results now follow from Definition 6.3 and

1. Definition 2.14 together with the observation under the third point of Example 6.4 for case (1).
2. Definition 3.2 together with the observation under the fourth point of Example 6.4 for case (2). □

So, indeed, *set- and partition-based* measure functions are finitely measuring for languages inherent in a finite program and goal. Algorithm 6.1 will certainly terminate when using them. Their common characteristic, guaranteeing these results, is the fact that all goals suitable for such a measure function have essentially the *same measured argument set* under that measure function. Example 6.5 already shows that this is not always the case for *cco-partition-based* measure functions. And indeed, Example 6.8 demonstrates that such measure functions are, in general, *not* finitely measuring. Of course, safely combining their use with some form of "equal weight but nonvariant" unfolding remains possible. It suffices to focus on the selected atom for the variant test. Alternatively, a specially tuned version of the "measured part" notion, to some extent also incorporating context information, can probably be developed. However, we will not devote a detailed study to this issue in the present paper. Rather, we shift our attention to automation of the "safe" cases in the next subsection.

6.3. Issues in Automation

Basically, it is very straightforward to adapt Algorithm 3.2 for fully automatic partition-based unfolding along the lines of the previous subsection.

Algorithm 6.2. We add one more item to the list of conditions enabling the selection of an atom for unfolding:

- *(G', j) is the direct covering ancestor of (G, i) and $|.|_{p,O_p}$ is $(R(G', j), p(t_1, \ldots, t_n))$-stable and $\neg\exists(G'', k) \in D : [(G'', k)$ covers $(G, i) \wedge |.|_{p,O_p}$ is $(R(G'', k), p(t_1, \ldots, t_n))$-stable $\wedge$ $(G'', k) \sim_{|.|_{p,O_p}} (G, i)]$.*

The rest of Algorithm 3.2 remains unchanged (except for one detail addressed below).

Example 6.9. Algorithm 6.2, applied to the program and query in Example 6.1, produces the SLD-tree depicted in Figure 10. Throughout the whole unfolding process, O_{reach} is never changed, and keeps its initial value: $(\{1, 2\})$. In general, it is obvious that no partition-based measure, as introduced in Section 3, ever has potential in a datalog context.

It is not difficult to realize that Algorithm 6.2 still terminates, and builds a finite, subsetwise nearly founded SLD-tree. One detail remains to be settled. Indeed, the enhanced list of conditions for atom selection *no longer contains mutually exclusive cases*: $|.|_{p,O_p}$ can, at the same time, be stable and have potential. Let us once more look at Example 6.2. The measure function $|.|_{rev,(\{1,2,3\})}$ is stable all along the top three goal nodes of the tree in Figure 9. Moreover, the *rev* atoms are not variants of each other. Therefore, Algorithm 6.2 allows unfolding without changing the measure function to $|.|_{rev,(\{1,2\},\{3\})}$, as would be done by Algorithm 3.2. (Note, however, that both terminate unfolding at node (*).) Such a behavior might be considered in conflict with our basic philosophy. It can easily be avoided by imposing a priority among the unfoldability conditions: first try a refinement; only if that fails, try unfolding on the "equal weight but nonvariant" basis.

A second point, and an additional motive for imposing the just mentioned priority, is the *potential inefficiency of the nonvariant check.* Indeed, the latter does require searching through a list of (equal weight) covering ancestors. So, the linearity property, established in Section 3.4, is lost. It might therefore be a good idea to restrict as much as possible the cases in which such scans are undertaken. Observe, however, that the implementation technique proposed in Section 3.4 provides excellent support for scanning chains of covering ancestors, if so desired. Reconsider Figure 2. If we select, e.g., the left p-atom in node (5), the covering ancestors can easily be spotted. The atom is annotated with the list [4, 2, 3] (remember that the first element of these lists refers to p). So, node (4) is the direct covering ancestor. Its selected p-atom has annotation [1, 2, 3]. This means that the next covering ancestor is node (1), and the "_" on the first position of its annotation list marks the top of the chain.

Summarizing, we can state that an integration of variant checking with *partition*-based unfolding is relatively straightforward, even in a fully automatic context. And, since the measured part for these measure functions invariably coincides with the whole selected atom, the full generality of the development in the previous subsection is, in this restricted context, probably somewhat of an overkill. Some open issues requiring further (experimental) research are:

- How big is the gain in sensible unfolding capacity outside a strict datalog context?
- How severe is the mentioned efficiency problem?

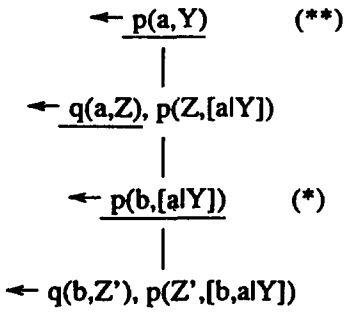

FIGURE 12. Combining set-based and nonvariant unfolding.

Next, we briefly address *set*-based unfolding. Since, now, part of a selected atom can be "disregarded," our integration enterprise involves some more subtle issues. Consider the following example:

Example 6.10.

$p(X,Y) \leftarrow q(X,Z), p(Z,[X \mid Y])$
$q(a,b) \leftarrow$

Part of an SLD-derivation for $\leftarrow p(a,Y)$ can be found in Figure 12.

In node $(*)$, we would like to be able to carry out the indicated unfolding step. This is possible with combined unfolding using $|.|_{p,\{1\}}$ as a measure function. However, setting out with the initial $|.|_{p,\{1,2\}}$, and adapting the notion of tight decreasing refinement to the context of set-based measures in the obvious way, the above measure function is *not* a $(**,*)$-*decreasing* refinement of $|.|_{p,\{1,2\}}$. In fact, there is none.

So, we need to revise the automatic unfolding algorithm in such a way that measure functions are refined into *nonincreasing* ones, possibly leading to useful stable unfolding on the basis of the newly added nonvariant condition. Doing so *can make set-based unfolding more powerful than partition-based* in some cases, thus outdating our conjecture at the end of Section 3.3. Of course, blending the two approaches is possible, through *partitioning subsets* of a selected atom's argument positions.

Finally, it can be mentioned that [22] contains some preliminary experimental data on partial deduction with a combined "set-based weights and nonvariant" unfolding strategy. (See also [33].) Tests were performed on (among others) the seven benchmark programs and queries proposed in [27]. Various criteria were used to govern unfolding, such as:

- standard set-based weights
- combined set-based weights and nonvariant checking
- plain nonvariant checking as described in, e.g., [2]
- the similar, but more restrictive, noncommon-instance checking
- weight-based unfolding, only for determinate goals.

A detailed discussion of the results leads too far here, but some preliminary conclusions can be included:

- Not surprisingly, the bare weight method performed badly on datalog programs. But the combined method unfolds very well.
- It is important to compare a goal only with *proper* (see Definition 2.8) ancestor goals. The basic nonvariant and noncommon-instance rules suffered very heavily from lacking this feature.
- Not having a common instance with an atom selected earlier seems to be too restrictive a condition, even when only proper ancestors are considered.
- The nonvariant rule, on the other hand, occasionally unfolds too deeply.
- In some cases, imposing the determinate-only condition was being too cautious: the resulting transformed programs were much less efficient than those obtained without this restriction.
- Finally, disregarding the determinism criterion, weight based unfolding (combined with nonvariant checking) in some cases showed the first signs of code explosion. Nevertheless, run-time performance of the resulting *compiled* code was very satisfactory, repeatedly being much better than that of any alternative, and never much worse.

More experimentation and analysis are required to gain a better insight into these and related topics. For further details, also on issues involved in the overall partial deduction algorithms used, we refer to [22] and [33].

7. DISCUSSION AND CONCLUSION

In this paper, we have extended earlier work on finite unfolding by Bruynooghe and ourselves. We have slightly adapted the basic framework built in [8] and shown various instances of it, capable of dealing with some advanced unfolding issues. We have consistently avoided ad hoc solutions as much as possible, always trying to concentrate on meaningful properties of the task at hand. Particular emphasis was put on the development of fully automatic unfolding algorithms, requiring no user assistance apart from providing a program and a query to be unfolded. Moreover, we have opted for a general, highly formalized presentation, trusting that this choice provides better chances for uncovering common underlying principles. In this way, we believe that we have enhanced the basic framework with a good understanding of issues involved in automatic, maximal, finite unfolding based on structural properties of the considered programs and goals. Finally, we have shown that weight-based unfolding as presented in Sections 2–5 enjoys good complexity properties, its execution effort being linear in the size of the generated SLD-tree when a clever implementation scheme is used.

A first major part of our work concentrated on methods relying on structure-based weights assigned to arguments upon which lexicographic priorities were imposed. Much of the inspiration for this approach came from work on termination of rewrite systems (see, e.g., [11]). The possible use of lexicographic well-founded orderings to control the unfolding of logic programs is already mentioned in [15]. To the best of our knowledge, however, our work is the first to carry out complete formal developments and present concrete algorithms based on this idea. We also believe that the work in Section 4 is the first to present concrete algorithms basing unfolding decisions in logic programs not solely on the shape of a goal's selected

atom, but also on contextual information in the rest of the goal. We have shown how our techniques can deal with coroutining and, to a certain extent, with back-propagation of variable instantiations. To be sure, a number of challenging issues remain as subjects for further research; we refer to Sections 4.3 and 4.4 for more detailed comments.

An important contribution of our work is the completely automatic search for optimal measure functions incorporated in our algorithms. Not only have we presented detailed formalizations showing how this search can be performed within specific classes of measure functions, but the common principles underlying various, increasingly complex algorithms in this and previous papers were identified in Section 5. As a result, Algorithm 5.1 and Theorem 5.1 can henceforth serve as a template and standard for discussion of automatic weight-based unfolding.

Next, in Section 6, we have integrated into our framework the well-known heuristic that an atom can be unfolded if it is not a variant of one already unfolded in the same derivation. As a standalone method, this heuristic does not provide safe unfolding. Combined approaches guarantee termination when specific conditions are fulfilled. Our work can be extended to reach more generally satisfied conditions for termination, or specific classes of instances of the framework can be studied in further detail, thus perhaps establishing termination in spite of the fact that the above-mentioned general conditions are not satisfied. Further experimental work is needed to assess unfolding power in complex cases, comparing different instances of the framework. Another issue that merits further attention is the performance characteristics of integrated methods. We already pointed out that the above-mentioned linearity property is lost. It may be that this has little practical consequences, but reestablishing linearity would certainly be of interest. An attempt in this direction might start from an idea mentioned in [5], where it is suggested that some order could be inferred among constants in a datalog program on the basis of the program's atom dependency graph. Such an off-line program analysis might enable the definition of a measure function again just requiring a comparison with the direct covering ancestor to decide on unfolding. Whether this idea actually is valid, is as yet unclear to us. More generally, it seems very likely that various sophisticated off-line analysis techniques might provide useful supporting information for on-line unfolding as studied in this paper.

Finally, work is in progress to extend our methods for automatic unfolding with a capability to *focus on subarguments.* Such a feature allows further advances in unfolding, e.g., of meta-interpreters. It requires a considerable additional formal apparatus, providing for overall refinements of $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$ pairs, including possible changes in the $R_0, R_1, \ldots, R_N$ partition. Descriptions of our current results on this issue, with formal details, preliminary algorithms, and examples, can be found in [34] and [33].

Further, we wish to point out that, in the formulation of the above algorithms, we have consistently reduced indeterminism to a minimum: summing argument weights per component in partitions, choosing the leftmost unfoldable atom, imposing the choice of one particular decreasing refinement, limiting the range of possible measure function switches through a "narrow" definition of the refinement concept. Only the choice of a nonterminated derivation, candidate for extension is left open. We have not studied in detail whether some of these decisions sometimes reduce unfolding potential. However, we conjecture that this will hardly ever be

the case. We therefore preferred to formulate algorithms in such a way that their implementation requires a minimum of extra design decisions. (On the other hand, the development of the basic framework and the generic treatment in Section 5 have both been kept very general.)

It has been mentioned before, but we can perhaps briefly repeat, that [37] presents a sound and complete method for partial deduction with unfolding using set-based measure functions. Moreover, [22] contains the results of some experiments, comparing the performance of different unfolding strategies and partial deduction techniques on a collection of benchmark programs. A brief preliminary discussion was included at the end of Section 6. Some further comments can also be found in [37]. Finally, a complete, detailed presentation of our work on the control of unfolding in partial deduction is offered in Part II of [33].

Next, [8] extensively discusses the relation between work on finite unfolding and termination analysis of logic programs. We see no need to repeat that discussion here. Let us simply add that [10] presents an extensive overview of the work in the latter field. Also mentioned in [8] is the open research issue of uncovering the relation between criteria for termination of unfolding during partial deduction and methods used to ensure termination in related program transformation techniques (see, e.g., [6], [20]). In this context, [41] also provides interesting study material.

Let us now discuss in somewhat further detail the unfolding strategies proposed in [45] (already briefly mentioned in Section 3.4). Translated into our framework, when an atom A in a goal G is considered for unfolding, it is compared with the selected atoms in G's proper ancestors. For each such atom A', a comparison $may\_loop(A', A)$ is executed. If this test succeeds at least a given number of times, unfolding is prohibited. Various possible instances for *may_loop* are listed:

- As mentioned before, the simplest test compares the two predicate symbols.
- A slightly more complicated test involves comparing functors. It succeeds if A and A' have predicate symbols of equal arity and all corresponding arguments contain the same (top level) functor. Since [45] addresses full Prolog, for a given program P and atom A, an SLD-tree might contain infinitely many different (dynamically created) functors. A distinction is therefore imposed between static and dynamic function symbols, the latter all considered identical in the *may_loop* test.
- A (more) natural extension of the previous approach considers complete structures to a given depth.
- Another instance demands that the total argument size should shrink, where this size is determined by applying a function like the one in Definition 2.13 to both atoms, considered as terms, also counting variable and constant symbols.
- Instead of all arguments, a single one may be used, or various arguments considered separately.
- Combining structure and size tests is also possible. The resulting *may_loop* has actually been used to control unfolding in most of the experiments described in [45] and [44]. Apart from a special treatment of dynamic functors, it checks whether two atoms are variants to a certain depth. If so, then the test succeeds if all subterms below that depth in the second have an equal or larger term size than corresponding subterms in the first. Using one fixed comparison strategy throughout its operation, this method allows to deal with

datalog, and offers a possibility to consider subargument evolution. Context atoms, however, are not taken into account, and entire ancestor goal stacks have to be inspected. Finally, two depth bounds (the structure depth for *may_loop* and the repetition parameter that determines the number of times *may_loop* is allowed to succeed) are to be chosen in advance and remain fixed during execution.

Finally, [3] should also be mentioned as recent, rather closely related work. Loop checks in logic programming are briefly discussed. They are classified as sound (keeping all solutions) and/or complete (removing all loops). It is argued that in the context of unfolding for partial deduction, sound loop checks can be helpful to improve the performance of partially deduced programs, but complete loop checks are essential to terminate unfolding. It is clear that all (terminating) unfolding algorithms above incorporate a complete loop check. Next, a detailed formalization of the various loop checking strategies proposed in [43] is included. Finally, [7] is discussed and its content briefly compared with the formalized [43] methods. It is stated that its basic framework might be of theoretical interest as a standard for complete loop checks, and natural instances should be easily implementable. We feel that the present paper confirms both conjectures.

We are grateful to Maurice Bruynooghe, John Gallagher, Dan Sahlin, and anonymous referees who commented on our work and provided valuable suggestions for its improvement.

REFERENCES

1. Benkerimi, K. and Lloyd, J. W., A Procedure for the Partial Evaluation of Logic Programs, Technical Report TR-89-04, Department of Computer Science, University of Bristol, U.K., May 1989.
2. Benkerimi, K. and Lloyd, J. W., A Partial Evaluation Procedure for Logic Programs, in: S. Debray and M. Hermenegildo (eds.), *Proc. NACLP'90*, Austin, TX, Oct. 1990, MIT Press, pp. 343–358.
3. Bol, R., Loop Checking in Partial Deduction, *J. Logic Programming* 16(1&2):25–46 (1993).
4. Bol, R. N., Apt, K. R., and Klop, J. W., An Analysis of Loop Checking Mechanisms for Logic Programs, *Theoretical Comput. Sci.* 86(1):35–79 (1991).
5. Brodsky, A. and Sagiv, Y., Inference of Monotonicity Constraints in Datalog Programs, in: *Proc. PODS'89*, Philadelphia, PA, Mar. 1989, ACM, pp. 190–199.
6. Bruynooghe, M., De Schreye, D., and Krekels, B., Compiling Control, *J. Logic Programming* 6(1&2):135–162 (1989).
7. Bruynooghe, M., De Schreye, D., and Martens, B., A General Criterion for Avoiding Infinite Unfolding During Partial Deduction of Logic Programs, in: V. Saraswat and K. Ueda (eds.), *Proc. ILPS'91*, San Diego, CA, Oct. 1991, MIT Press, pp. 117–131.
8. Bruynooghe, M., De Schreye, D., and Martens, B., A General Criterion for Avoiding Infinite Unfolding During Partial Deduction, *New Generation Computing* 11(1):47–79 (1992).
9. Cousot, P. and Cousot, R., Abstract Interpretation and Application to Logic Programs, *J. Logic Programming* 13(2&3):103–179 (1992).
10. De Schreye, D. and Decorte, S., Termination of Logic Programs: The Never-Ending Story, Technical Report CW182, Departement Computerwetenschappen, K.U. Leuven, Belgium, Oct. 1993.

11. Dershowitz, N. Termination of Rewriting, *J. Symbolic Computation* 3(1&2):69–115 (1987). Corrigendum in 4(3):409–410.
12. Dershowitz, N. and Jouannaud, J.-P., Rewrite Systems, in: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science, Vol. B*, Elsevier, MIT Press, 1990, pp. 243–320.
13. Fitting, M., *First-Order Logic and Automated Theorem Proving*, Springer-Verlag, 1990.
14. Floyd, R. W., Assigning Meaning to Programs, in: *Proc. Symp. in Applied Math., 19*, Providence, RI, 1967, Amer. Math. Soc., pp. 19–32.
15. Fujita, H. and Furukawa, H., A Self-Applicable Partial Evaluator and Its Use in Incremental Compilation, *New Generation Computing* 6(2&3):91–118 (1988).
16. Fuller, D. A. and Abramsky, S., Mixed Computation of Prolog Programs, *New Generation Computing* 6(2&3):119–141 (1988).
17. Gallagher, J., Transforming Logic Programs by Specialising Interpreters, in: *Proc. ECAI'86*, 1986, pp. 109–122.
18. Gallagher, J., A System for Specialising Logic Programs, Technical Report TR-91-32, Department of Computer Science, University of Bristol, U.K., Nov. 1991.
19. Gallagher, J., Specialisation of Logic Programs: A Tutorial, in: *Proc. PEPM'93, ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, June 1993, ACM Press, pp. 88–98.
20. Gallagher, J. and Bruynooghe, M., The Derivation of an Algorithm for Program Specialisation, in: D. H. D. Warren and P. Szeredi (eds.), *Proc. ICLP'90*, Jerusalem, June 1990, MIT Press, pp. 732–746. Revised version in *New Generation Computing* 9(3&4):305–333.
21. Gurr, C. A., *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*, Ph.D. thesis, Department of Computer Science, University of Bristol, U.K., Jan. 1994.
22. Horváth, T., Experiments in Partial Deduction, Master's thesis, Departement Computerwetenschappen, K.U. Leuven, Leuven, Belgium, July 1993.
23. Jones, N. D., Gomard, C. K., and Sestoft, P., *Partial Evaluation and Automatic Program Generation*, Prentice-Hall, 1993.
24. Komorowski, J., A Specification of an Astract Prolog Machine and Its Application to Partial Evaluation, Technical Report LSST69, Linköping University, 1981.
25. Komorowski, J., Guest Editor's Introduction, *J. Logic Programming* 16(1&2):1–3 (1993).
26. Komorowski, J. (ed.), *J. Logic Programming* 16(1&2), Special Issue on Partial Deduction (1993).
27. Lam, J. K. K. and Kusalik, A. J., A Comparative Analysis of Partial Deductors for Pure Prolog, Technical Report, Department of Computational Science, University of Saskatchewan, Saskatoon, Sask., Canada, May 1990. Revised Apr. 1991.
28. Leuschel, M., Self-Applicable Partial Evaluation in Prolog, Master's thesis, Departement Computerwetenschappen, K.U. Leuven, Leuven, Belgium, Sept. 1993.
29. Levi, G. and Sardu, G., Partial Evaluation of Metaprograms in a Multiple Worlds Logic Language, *New Generation Computing* 6(2&3):227–247 (1988).
30. Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, 1987.
31. Lloyd, J. W. and Shepherdson, J. C., Partial Evaluation in Logic Programming, *J. Logic Programming* 11(3&4):217–242 (1991).
32. Manna, Z. and Ness, S., On the Termination of Markov Algorithms, in: *Proc. 3rd Hawaii Int. Conf. on Syst. Sci.*, Honolulu, HI, 1970, pp. 784–792.
33. Martens, B., On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming, Ph.D. thesis, Departement Computerwetenschappen, K.U. Leuven, Belgium, Feb. 1994.

34. Martens, B. and De Schreye, D., Advanced Techniques in Finite Unfolding, Technical Report CW180, Departement Computerwetenschappen, K.U. Leuven, Belgium, Oct. 1993.
35. Martens, B. and De Schreye, D., Some Further Issues in Finite Unfolding (Abstract), in: Y. Deville (ed.), *Proc. LOPSTR'93*, Louvain-la-Neuve, Belgium, 1994, Springer-Verlag, Workshops in Computing Series, pp. 159–161.
36. Martens, B., De Schreye, D., and Bruynooghe, M., Sound and Complete Partial Deduction with Unfolding Based on Well-Founded Measures, in: ICOT (ed.), *Proc. FGCS'92*, Tokyo, June 1992, Omsha Ltd., pp. 473–480.
37. Martens, B., De Schreye, D., and Horváth, T., Sound and Complete Partial Deduction with Unfolding Based on Well-Founded Measures, *Theoretical Comput. Sc.* 122(1–2):97–117 (1994).
38. Martens, B. and Gallagher, J., Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance, in: L. Sterling (ed.), *Proc. ICLP'95*, Shonan Village Center, Japan, June 1995, MIT Press, pp. 597–611.
39. Mogensen, T. and Bondorf, A., Logimix: A Self-Applicable Partial Evaluator for Prolog, Technical Report, DIKU, Department of Computer Science, University of Copenhagen, Denmark, Apr. 1992.
40. Prestwich, S., Online Partial Deduction of Large Programs, in: *Proc. PEPM'93*, Copenhagen, Denmark, June 1993, ACM, pp. 111–118.
41. Proietti, M. and Pettorossi, A., The Loop Absorption and the Generalization Strategies for the Development of Logic Programs and Partial Deduction, *J. Logic Programming* 16(1&2):123–161 (1993).
42. Safra, S. and Shapiro, E., Meta Interpreters for Real, in: H.-J. Kugler (ed.), *Information Processing 86*, 1986, pp. 271–278.
43. Sahlin, D., The Mixtus Approach to Automatic Partial Evaluation of Full Prolog, in: S. Debray and M. Hermenegildo (eds.), *Proc. NACLP'90*, Austin, TX, Oct. 1990, MIT Press, pp. 377–398.
44. Sahlin, D., An Automatic Partial Evaluator for Full Prolog, Ph.D. thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden, 1991.
45. Sahlin, D., Mixtus: An Automatic Partial Evaluator for Full Prolog, *New Generation Computing* 12(1):7–51 (1993).
46. Sterling, L. and Beer, R. D., Meta Interpreters for Expert System Construction, *J. Logic Programming* 6(1&2):163–178 (1989).
47. Takeuchi, A. and Furukawa, K., Partial Evaluation of Prolog Programs and Its Application to Metaprogramming, in: H.-J. Kugler (ed.), *Information Processing 86*, 1986, pp. 415–420.
48. Venken, R., A Prolog Meta Interpreter for Partial Evaluation and Its Application to Source to Source Transformation and Query Optimization, in: T. O'Shea (ed.), *Advances in Artificial Intelligence, Proc. ECAI'84*, North-Holland, 1984, pp. 347–356.
49. Venken, R. and Demoen, B., A Partial Evaluation System for Prolog: Some Practical Considerations, *New Generation Computing* 6(2&3):279–290 (1988).